Memotech MTX Series

VDP DISCOVERED

(Advanced Reference Manual)

by

AFW Software

M
S
X

M
S
X

The Tatung Einstein

CONTENTS

1.0 Introduction

1.1 Overview

Between 1983 and 1984, three microcomputers sharing almost identical hardware architecture , ie same CPU & graphics chip VDP), hit the streets. They were the U.K. based Memotech MTX and the Tatung Einstein and the mighty Japanese MSX computers ,like the Sony Hibit. All three machines (referred to as the VDP Compendium) were priced at around ` 300 - ` 500. The VDP Compendium was offering superb technical specification , but at a price out of the reach of many potential users. This allowed the Sinclair ZX Spectrum & the Commodore 64 establish themselves as the main Competitors in the low cost games and educational markets.

Following on from this the much awaited new Sinclair computer ,Sinclair Ql, and the arrival of the Amstrad CPC 464 with its built-in tape recorder and monitor, both priced at about `400 , destroyed any chances the VDP compendium had. Even when all three machines eventually did cut their prices, thus making them far more attractive , the lack of software compared with the Spectrum and Commodore 64 , was a serious drawback , even though the software produced was far superior.

The next step was the production of a number of excellent ZX Spectrum, software emulators. The initial emulators , just allowed ZX BASIC programs to run on the VDP Compendium. The next stage of emulators allowed specific commercial games to run. However, the emulator didn't allow the user to go out and buy ZX Spectrum software, load it in and run it. First of all they had to buy a special tape that was setup to run at least 20 games, thus increasing the cost and you were not guaranteed that the software you wanted was available on a special tape. [NB: The arrival of the MGT sam coupe computer does allow proper ZX Spectrum emulation without compromise].

Finally, Version 2's hit the streets , ie the Einstein 256 or the MSX 2 or the Memotech MTX series 2 , offering far better performance and specification at competitive prices,but still didn't woo the public who wanted games software ,NOW . Even the arrival of the 16-bit Atari ST and CBM Amiga , have taken 4 or 5 years to get a foothold in the games market and even then its negligible compared with the ZX Spectrum and CBM 64.

The VDP compendium machines have niche markets and because of their technical excellence , construction and good quality software available will remain for some time to come.Machines like the Memotech and Einstein and MSX 2 offer the computer user the opportunity to upgrade to CPM , the 8 - bit software business standard. Such classics as Dbase II or Wordstar or Supercalc are now cheap and readily available.

1.2 Objectives/Aims

Originally , this computer manual was aimed at Memotech MTX owners. However , as I looked at other Z80 based systems, it became apparent that the Einstein and MSX computers shared a common architecture with the Memotech MTX series - Z80 CPU , VDP graphics chip , IN / OUT mapped hardware devices , CPM expansion , similar sound capabilities (3 channel) .

Rather than a complete rewrite , I decided to leave the bulk of the text as is and add two short appendixes to help users of the MSX and Einstein understand and use the information within the manual.

"VDP Discovered" is a comprehensive technical manual on the workings of the Texas Instruments VDP graphics processor. However, the interactive nature of graphics with joysticks , keyboard, screen dumps and RAM / ROM , required an expansive treatment to include such overlap. Certainly , not for the novice, but anyone who is interested in Z80 machine code and writing software whether utilities or games or ??? , shouldn't be without this excellent source of information.

1.3 The Listings

Throughout this manual the prefix # is used to signify that the number following it is a hexadecimal number, ie #9000. However , some other computers/assemblers use the ampersand sign to prefix hex numbers, ie &A000. Some other systems use the postfix "H" to indicate a hex number, ie DFFFH. Use the nomenclature that your system uses.

All listings in this book have been designed to be well structured, informative and above all modular. The reason for modularity is that it means that many of the subroutines designed for one listing can be incorporated into other program listings. A follow up manual called POWER GRAPHICS , extracts many of the subroutines , enhances them and adds new ones, like DRAWLINE , DRAWBOX , FILLBOX, TRIANGLE ,etc. This manual is a Z80 graphics (VDP) library. Ideal companion for all those thinking of writing programs, like CAD,DTP,etc.

All listings were tested on a MTX 512 tape system, and a FDX/SDX disc system. All listings were as written , see examples within, ie used the Memotech built-in Z80 assembler and MTX BASIC (with Disc BASIC,if FDX/SDX disc system used).

However,for the professional Z80 programmer , who prefers too write and develop utilities/programs ,in CPM mode,then you'll also find the current listings compatible . The listings were written using Newword wordprocessor in non-document mode (or ASCII mode) to generate the Z80 assembly source text , and compiled (assembled) into Z80 machine code or object code using Hisoft's DEVPAC80 (v2) Z80 compiler . The procedure for generating and running the object code is as follows:

Save the source text to disc as "fname.gen"
Compile the source text into an object file or COM file, ie "fname.com" using DEVPAC80 .
Load the Memotech with Disc Basic: FDXB 40 or FDXB07 40 <RET>

This will return you to MTX Disc Basic Mode. Now Load the compiled Z80 object code with:

```
DISC (USER) READ "fname.COM",33031
```

Now type in the BASIC part of the listing, replacing the occurrences of GOSUB <line number> with RAND USR(33031).

MSX and Tatung Einstein users please refer to the appropriate appendix for the method of loading , saving and running the enclosed listings.

Note that the SOURCE file is the Z80 assembly language text file. Whereas,the OBJECT file is the compiled version of the source file. The object file is a file in binary form for the computer to read.

1.4 The Controller Devices

The Memotech MTX series uses dedicated industry standard IN/OUT mapped hardware devices (controllers) for handling the computers sound ,graphics , communication and Disc filing capabilities. This approach was in preference of the in-house custom designed hardware devices. The use of custom chips allows one chip to handle a number of different functions,ie like sound and graphics and reduce the overall chip count and the size of the computer. This type of approach is certainly cost effective in the long term but it was extremely expensive to design and debug in 1983 - 1984, when the Memotech MTX series was being developed.

However, better design software and cheaper and better manufacturing technology have greatly improved the speed and cost effectiveness of this approach as Acorn have found with its ARM series of RISC CPUs and controllers. Therefore, Memotech opted to use the industry standard devices because:

a) They were standard hardware devices,there would be a plentiful supply as more than one manufacturer. Also increased competition between suppliers brings the prices of the devices down.

b) Circuit design would be easier because of increased literature documentation.

c) Other computers using the same architecture and devices ,like the MSX and Einstein machines, should have enticed software houses to write for these machines as very little re-writing would have been necessary,because they used the same graphics chip and the architecture.

Chapters 2 to 10 are devoted to explaining how these devices interface with the CPU and the Z80 programmer.

1.5 MTX Memory Map Architecture

The Memotech MTX series has deen designed to operate as a ROM/RAM based memory system , as in normal MTX mode,see Figure 1-1. However, for some purposes,as for CPM mode, the memory map must also be configured as a RAM only architecture. This latter architecture is very similar to the memory map in Figure 1-1,execpt that the 7 pages of ROM (16k) are used as 16k of RAM. All this will become a lot clearer in a moment or two but first study the Memory Map below.

Page: Num	**Common ROM**	**Application ROM**	**RAM 2 BLOCK**	**RAM 1 BLOCK**	**Common RAM**
0	SYS-A	SYS-Basic	512	500/512	Common
1	common	SYS-C	exp	512	to all
2	to all	USER ROM	exp	exp	16 pages
3	7 pages	USER ROM	exp	exp	of RAM
4	of ROM.	USER ROM	exp	exp	and is
5	It is	FDX EPROM	exp	exp	available
6	readily	NODE EPROM	exp	exp	all the
7	available	KBD EPROM	exp	exp	time,no
8	/ / / /	/ / / / / /	exp	exp	matter
9	/ / / /	/ / / / / /	exp	exp	what PAG
10	/ / / /	/ / / / / /	exp	exp	your on.
11	/ / / /	/ / / / / /	exp	exp	This PAG
12	/ / / /	/ / / / / /	exp	exp	has the
13	/ / / /	/ / / / / /	exp	exp	system
14	/ / / /	/ / / / / /	exp	exp	& BASIC
15	/ / / /	/ / / / / /	exp	exp	variables
SIZE	**8k**	**8k**	**16k**	**16k**	**16k**

where:-

exp	= 16k RAM expansion.
500	= This label indicates that this RAM block is available on the MTX 500 series.
512	= as for 500,except available on the MTX 512.
SYS-A	= This ROM holds the startup code and the MTXOS.
SYS-B	= This ROM holds the BASIC Interpreter Code.
SYS-C	= This ROM holds the PANEL and other utility code.
USER	= A User ROM can be inserted here. One of these slots is used for the SDX EPROM.
FDX	= Holds the FDX bootstrap Code.
NODE	= This ROM is needed for Networking.
KBD	= This is the Custom KBD EPROM.
Common	= These Pages, whether RAM or ROM,have been linked so that they are available on all PAGES, see later.

Figure 1-1: The Memotech ROM/RAM Architecture as for MTX 512.

Let us take a closer look at the MTX mode architecture. There are two RAM blocks of RAM providing 16k per page. These two blocks of 16k provide the user with 32k of RAM per PAGE. This can be expanded to a maximum of 512k (16*32k). The user Ram is used to store BASIC and Z80 programs, and text. The user RAM is located between #4000 and #BFFF. The other 16k of RAM or Common RAM is located between #C000 and #FFFF. This area of RAM has been joined to all pages so that it is available on all pages. Its primary function is to hold the System variables and the variables and workspace needed to RUN BASIC programs held in the USER RAM area, ie arrays, variables, strings.

The 16k ROM block is subdivided into two 8k subblocks. Located at #0000 to #1FFF is the Common ROM, SYS-A. This ROM holds all the necessary code to set up the system variables ,etc and holds the gateways to the Graphics and Number crunching functions, ie RST 10 and RST 28 respectively. The reason for using Common ROM or RAM will become apparent in sections 1.6 and 1.7. The Application ROM subblock is located between #2000 and #3FFF. The Application ROM subblock as its name suggests holds the ROMs that we need,ie BASIC,PANEL ,networking and Disc Operations. However,we can also plug in other ROMS/EPROMS like PASCAL, Wordprocessor (NEWWORD), or even our own.

You may be wondering why we need so many 64k pages when surely it would be easier to increase the 64k length to say 128 or 512k etc. Well this is because of the limitations imposed on us by the Z80 CPU. The Z80 only provides 16 address lines. The number of address lines tell us how much memory the CPU can address or manage on one PAGE. The Z80 can address or access upto 2^16 or 64k only. The only way around this constraint is to provide more than one 64k block and to PAGE Switch as the need becomes necessary. More on this later.

1.6 <u>ROM Page Switching</u>

The computers Memory (RAM and ROM) is also thought of as a hardware device and as such is mapped as an IN/OUT device. This means that the device is accessed by the Z80 CPU via one of its eight 8-bit INPUT or OUTPUT lines,of which there are 256 respectively. OUTPUT port or line #00,has been reserved for changing the ROM or RAM page. This technique is known as BANK switching. On the MTX series one byte is responsible for controlling which BANK of 64k is used and whether the device is mapped as a ROM/RAM architecture or just RAM architecture. Figure 1-2 describes the composition of this one byte.

Bit :	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	MODE	R2	R1	R0	P3	P2	P1	P0

Figure 1-2: The byte that sets up the Memory architecture and selects the 64k Page.

Bit 7 (MODE) when off (0) means a RAM/ROM architecture has been selected and if bit 7 is on (1) then a RAM architecture has been selected. At this stage,we will ignore P3 to P0, until section 1.7 ,but suffice to say, these 4 bits determine which BANK of RAM we are in. The remainder of this section deals with the 3 bits which determine which ROM page has been selected.

In BASIC a ROM page can be accessed or selected using the command ROM n ,where n= 0 to 7. What this command does is to switch in the appropriate ROM/EPROM ,in the desired slot and executes a CALL to #2010 on the selected ROM/EPROM and this RUNs the application, ie PASCAL or NEWWORD or the SDX disc bootstrap. Bits 4 - 6, are used to select the desired ROM page.,ie 2^3 = 8   permutations or 8 ROM applications. Selecting a ROM page from Z80 assembly language is a straight forward excercise,as shown in Listing 1-1.

Listing 1-1: Moving Code from ROM page 0 to RAM Page 0 at #9000, so that the MTX keyboard scan code can be disassembled with the PANEL.

```
90 CODE
           PUSH HL           ;
           LD A,(#FAD2)      ; SYSTEM VARIABLE #FAD2 HOLDS THE
           LD (OLDPAGE),A    ; CURRENT PAGE CONFIGURATION. SAVE
           AND #8F           ; IT AND MASK WITH 10001111. THIS
                             ; WILL LEAVE THE MODE AND RAM PAGE
           LD L,A            ; UNCHANGED.
           LD A,(ROMNUM)     ; A = ROM PAGE NUMBER.
           SLA A             ; *2
           SLA A             ; *4
           SLA A             ; *8
           SLA A             ; ROM PAGE * 16 = ROM NUM IN TERMS
                             ; OF BITS 4,5 & 6.
           ADD A,L           ; THE NEW PAGE CONFIGURATION.
           LD (#FAD2),A      ; TELL MTXOS ABOUT THE CHANGE.
           OUT (#00),A       ; TELL THE Z80.
                             ; INSERT CALL PROGRAM. BUT, I CHOSE
                             ; A PRACTICAL EXAMPLE, see Chapter 9
           EX (SP),HL        ; SAVE THE STACK POSITION
           PUSH HL           ;
           LD HL,#2000       ; START OF ROM 0.
           LD DE,#9000       ; WHERE IN RAM TO MOVE IT TOO.
           LD BC,#1FFF       ; SIZE OF ROM (8K).
           LDIR              ; THE CODE IS MOVED TO RAM AT #9000
                             ; INSERT COMPLETED. RETURN TO OLD
           POP HL            ; PAGE CONFIGURATION.
           EX (SP),HL        ; DON'T FORGET TO RESTORE THE STACK
           LD A,(OLDPAGE)    ; GET OLD CONFIGURATION
           LD (#FAD2),A      ; TELL MTXOS
           OUT (#00),A       ; TELL Z80.
           POP HL            ;
           RET               ;
OLDPAGE:   DS 1              ; SAVE OLD PAGE CONFIGURATION.
ROMNUM:    DB #00            ; ROM PAGE 0 SELECTED.
```

Save as:

```
     SAVE "ROMSELTXT"                             (tape users)
     DISC (USER) SAVE "ROMSEL.TXT"                (disc users)
```

Reload and RUN <RET> ,the above code. When you are returned to the BASIC flashing cursor,type PANEL. Now list (L) the code from #9622. You should get:

```
LD A,#FB
LD (#FD7E),A
OUT (#05),A
IN A,(#05)
BIT 0,A
JR Z,#9636
```

The above code is part of the MTX's keyboard scan routines which is accessed from the common ROM at #0079,which then jumps to CALL #3618 on ROM Page 0 ,which we have just moved and can disassembly further if required. Note that,although we have moved the code from #3618 to #9618 ,all absolute address calls ,ie CALL #3622 or JP #3618 will remain as such but relative addresses will change,ie JR #3636 to #9636.

The above listing is straight forward enough and can be implemented for any ROM page. This process is common practice on the MTX as it's constantly switching pages especially when it needs to RUN BASIC listings. At swtich on, ROM Page 1 is switched in, check this by entering PANEL and Displaying #FAD2.

1.7 RAM Page Switching

Page switching is very much akin to ROM switching,except that up to 16 different pages are switchable to ,and thus 4 bits (P3,P2,P1,P0) are needed for selection,ie 2^4 = 16 permutations.At switch on,RAM Page 0 ,is selected. However, users with MTX 512s or RS128s or the new MTX Series 2 have the ability to switch in RAM from other pages. However,this is a bit tricker than for ROM page switching. The reason for this will become apparent.

When a program is RUN, an internal pointer keeps track of the next command to be executed. The commands are usually held in RAM ,ie BASIC programs or Z80 programs. However,when we start swapping the RAM page, what we are doing is keeping the pointer at the address for the next command,but because we have swapped RAM pages,the code that we were executing is on a different page and the system will crash,as it cannot execute the code to switch the code back.

This is however overcome by placing the code in the common RAM area,ie between #C000 and #FFFF. Because this RAM is common to all pages,the code will be available to all pages. Therefore there is no problem in the executing the next command even though we are switching pages. However,if you place your code in this area of RAM and if your are running it from within a BASIC program, then the position of the code will become important as BASIC variables may overwrite it. This will not be a problem in the next listing.

Listing 1-2: RAM Page Switching Example (not for MTX 500).

```
95 CODE

8007       PUSH HL          ;
8008       PUSH AF          ;
800B       LD A,(#FAD2)     ; SAVE OLD CONFIGURATION
800E       LD (#D030),A     ;
8010       AND 240          ; MASK WITH 11110000,ie KEEP THE
8011       LD L,A           ; CURRENT MODE AND ROM PAGE.
8014       LD A,(#D031)     ; A = THE NEW RAM PAGE NUMBER.
8015       ADD A,L          ; GET NEW PAGE CONFIGURATION.
8018       LD (#FAD2),A     ; TELL MTXOS
801A       OUT (#00),A      ; TELL Z80
801B       EX (SP),HL       ; SAVE STACK
801C       PUSH HL          ;
801F       LD A,(#9000)     ; READ THE VALUE AT #9000 ON PAGE 1
8022       LD (#D032),A     ; SAVE IT TO READ ON PAGE 0.
8023       POP HL           ; RESTORE STACK
8024       EX (SP),HL       ;
8027       LD A,(#D030)     ; RESTORE OLD CONFIGURATION.
802A       LD (#FAD2),A     ; TELL MTXOS
802C       OUT (#00),A      ; TELL Z80
802D       POP AF           ;
802E       POP HL           ;
802F       RET              ;
8030       DS 1             ; OLDPAGE
8031       DB #01           ; RAM PAGE NUMBER
8032       DS 1             ; RESULT
```

Now enter PANEL and move the code from 8007 to D007 ,ie

```
Move>8007
End> 8033
To>  D007
```

The above listing was written so that moving the code would involve little effort apart from adjusting the storage positions as these wouldn't have adjusted on moving as they are Absolute addresses. Now delete line 95 and type at line 10 RAND USR(53255) <RET>. Now type RUN <RET>. The code at #D007 would have been executed. Enter PANEL and check the contents of RESULT at #D032. This value should be CD. This is a very trival example but it does get the point across.

Finally,FDX owners, when you load the FDX Disc Basic and look at #FAD2 ,you will find that the value is #90 and not #10 as in MTX mode. #90 relates to 10010000 ,ie this is a RAM based system as opposed to the MTX ROM version. This is also the architecture used in CPM mode as this is a RAM mode.

2.0 **Graphics Technical Overview**

2.1 Definitions of VDP and VRAM

The Video Display Processor,VDP, as used in the Memotech MTX series,the Tatung Einstein and the Japanese MSX microcomputers is the Texas Instruments TMS 9918/28/29. The VDP is a dedicated graphics microprocessor with nine internal registers (eight of which are write only [R0-R7] and one Read only ststus register [R8]).

For a microprocessor to function properly,it must have access to Random Access memory,RAM. The 8-bit/16-bit addressing architecture of the Z80 CPU can address only 2^16 or 65536 bytes. Note that,the MTX can access upto 512k by bank switching-see chapter 1.0 .

However,as most computer operating systems,OS,and built in BASIC Interpreters can grab between 8-32k of RAM and then reserve another 8-20k for graphics; this doesn't leave the programmer much room to spare. The BBC B series are prime examples of this. The amount of RAM the OS reserves for graphics depends on the screen mode's colour & pixel resolution;the greater the pixel and/or colour resolution the more RAM required to store it.

On the other hand,the Memotech,MSX and Einstein machines have a special RAM block which is used only for graphics. This block of RAM is 2^14 or 16384 bytes long and is independent of the CPU's RAM. The special RAM is called Video RAM, VRAM. VRAM is used for storing all graphic,sprite and text patterns. Figure 2-1,illustrates the relationship between the VDP/VRAM and the CPU/RAM systems.

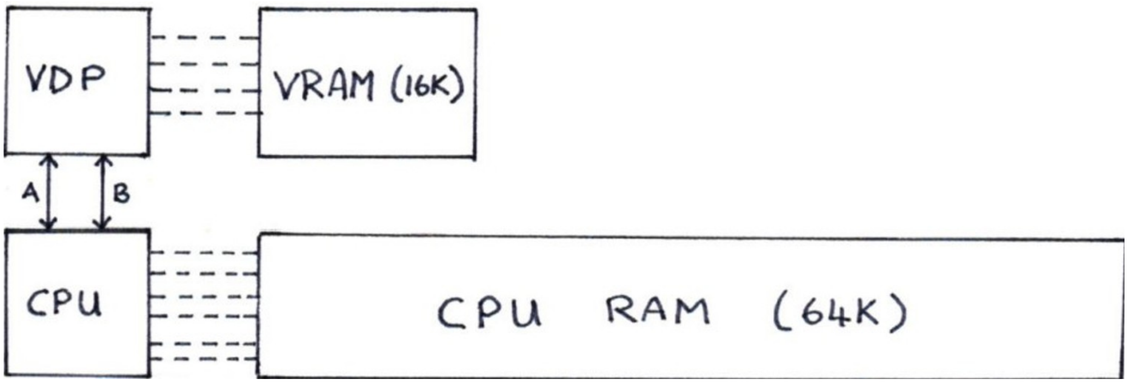


Figure 2-1: VRAM Memory Architecture.

Figure 2-1 is a IN - OUT mapped architecture, where the Z80 CPU communicates with the VDP via the READ (IN) & WRITE (OUT) data lines on the CPU,see chapter 10 on SOUND for a more detailed description. The advantages of a IN-OUT mapped dedicated graphics processor with its own private ram,VRAM,is threefold:

(i) Programming space in normal RAM is at a premium.
(ii) A dedicated graphics processor and its own RAM, will allow better and faster graphic manipulation as needed in arcade style games and particularly in animation.
(iii) The CPU can get on with the rest of the program without having to stop executing the program in order to update the screen map. This speeds up processing,which may be essential in number crunching.

2.2 CPU and VDP Communication

As already stated the VDP communicates with the host CPU via communication lines,ie Ports A & B,see figure 2-1. Port A is used for data transfers to and from the VDP & CPU,ie the moving (writing) of data to VRAM or reading data from VRAM,for example,reading the screen as required for screen dump printouts,see chapter 8.

Port B,is used to set the VDP addressing for either Reading (ie, status register) or Writing (ie, setting the screen pointer to a specific VRAM address). Once the VRAM address has been selected,we can send via PORT A, for instance, a sprite shape. Table 2-1 gives the actual port numbers for the computers under discussion.

Table 2-1: The ports used by the Memotech,MSX and Einstein Z80 microcomputers for VDP-CPU communication. where r/w =reading or writing data transfers.

Computer	**port A (data r/w)**	**port B (addressing)**
Einstein	#08	#09
Memotech	#01	#02
MSX	#98	99

2.3 Screen Modes and Resolutions

The TI TMS 9928/18/29 VDP provides the programmer with four built in screen modes: TEXT,Graphics I and II and Multicolour. The latter 3 screen modes,can display upto 32 hardware sprites. This allows the programmer to animate characters with ease-see chapter 7.0. The VDP has a dot resolution of 256x192 .

At this stage,it would be advantageous to skip the rest of this chapter and read chapters 3,4,5 & 6. These chapters will give you a better understanding of the functions of the VDP registers and how to use the VDP more effectively.

2.4 VDP Registers

As already stated,the VDP has 9 internal registers numbered 0 to 8. Registers 0-7 are write only and register 8 is the read only status register. The VDP requires three pieces of information to alter or set a register:

1. Which register needs setting or altering ?
2. Is it a READ [R8] or WRITE register [R0-R7]. ?
3. What value are we going to set or alter the register to ?

This information can be selected with only two bytes (a) the data byte and (b) the register/read/write byte. The data byte is a 8-bit data byte which is used in conjunction with Table 2-3,to give the VRAM addresses of specific areas like patterns,colour tables ,etc. The register/read/write byte has two functions,(i) it tells the VDP which register is selected and (ii) it tells the VDP whether to READ or WRITE to that register,see Table 2-2 for a description of this byte.

Table 2-2: VDP Register/READ/WRITE byte description.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RS	**R/W**	**0**	**0**	**RS3**	**RS2**	**RS1**	**RS0**
1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	1
1	1	0	0	0	0	1	0
1	1	0	0	0	0	1	1
1	1	0	0	0	1	0	0
1	1	0	0	0	1	0	1
1	1	0	0	0	1	1	0
1	1	0	0	0	1	1	1
1	0	0	0	1	0	0	0

where:

RS	=	Register Selected {1} NOT Selected {0}
R/W	=	READ {0} or WRITE {1}
RSn	=	register pattern.

Each of the 8 registers (see register patterns RS3-RS0) has a specific task ,like setting the colour or testing for sprite collision,etc. These functions are summarised in table 2-3 and a slightly more detailed description is given below for each of these functions. The initialisation of the VDP registers is covered later.

Table 2-3: Summary of the Functions of the VDP.

Reg No.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
#00	0	0	0	0	0	0	M3	EV
#01	1	BLK	IE	M1	M2	0	SIZE	MAG
#02	0	0	0	0	<- NAME TABLE BA -->			
#03	<------- COLOUR TABLE BASE ADDRESS ------->							
#04	0	0	0	0	<-PATTERN TABLE BA->			
#05	0	<---- SPRITE ATTRIBUTE TABLE BA ---->						
#06	0	0	0	0	0	<- S.P.T.B.A.->		
#07	<-- INK COLOUR -->				<---- PAPER COLOUR ---->			
#08	F	5S	C	<-- 5th SPRITE NUMBER --->				

All the entries above will be discussed in more detail in the next couple of pages. It is worth memorising this simple table as it is a valuable tool to know,especially when writing and debugging graphical programs.

2.4.1 Description of the VDP Function Table

Register 0:

bit 0: external video control. This bit is disabled at switch on {0}. It is used if you require to use an external VDP board.

bit 1: M3 is the pattern bit mode. This is used in conjunction with M2 & M1 (register 1),to determine the screen mode. See Table 2-4.

bits 2 to 7 are NOT USED and are set to Zero.

Table 2-4: VDP Screen Mode Select

Screen Mode	**Resolution**			**Mode Select**		
	Text	**Graphics**	**Colour**	**M1**	**M2**	**M3**
Graphics I	32x24	256x192	02xchar	0	0	0
Graphics II	32x24	256x192	16xchar	0	0	1
Text	40x24		02xscrn	0	1	0
Multicolour		64x 48	02xdot	1	0	0

where:

char	=	8x8 pixels or dots
scrn	=	the whole screen
dot	=	1x1 pixel.Multicolour mode a dot is 4x4 pixels.

Register 1:

bit 0: Sets Sprite Magnification; 0 = *1 ; 1 = *2.
bit 1: Sets the Sprite Size; 0 = 8x8 ; 1 = 16x16.

Table 2-5: Onscreen Sprite Size

bit 1	**bit 0**	**Sprite Size**	**Dot Size**
0	0	8 x 8	1 x 1
1	0	16 x 16	1 x 1
0	1	16 x 16	2 x 2
1	1	32 x 32	2 x 2

bit 2: Not used and set to zero.
bit 3: See Table 2-4.
bit 4: See Table 2-5.
bit 5: This is the VDP interrupt signal; 0=disable
       1=interrupt enabled.
bit 6: Blank Screen bit ; 0 = blank ; 1 = normal.
bit 7: Set to 1 on the MTX,MSX & Einstein for 16k Video.

Register 2:

Bits 0 - 3 (or lower nybble,LSN) are used to calculate the VRAM starting address of the Pattern Name Table .

```
 -----------------------------------------
  NAMEBASE = Register 2 * #400 (or 1024)
 -----------------------------------------
```

Since we are only using the lower 4 bits in the calculation ,the register 2 range is 0-15 and because we are multipling by 1k,we know that the NAME TABLE BASE or starting address is located on 1k boundries only ,ie if Register 2 = 10 ,then the NAMEBASE = 10 * 1024 = 10240 or 10k.

This Table is used to keep track of which patterns are on the screen at any time. In graphics I or II mode,this table is 768 bytes long giving a text resolution of 32x24. In text mode,this table is 960 bytes long giving a text resolution of 40x24.This table stores one byte identifiers,or ASCII numbers. Since the Pattern Name Table is mapped as either 40x24 (TEXT-960) or 32x24 (GRAPHICS-768), the ASCII number is stored at the desired cursor coordinates.This unique ASCII number relates to a particular pattern in the Pattern generator table,see register 4.

At switch on the MTXOS loads both the graphic and text generator patterns with the desired ASCII patterns. This process is only required once. When writing to the screen,all we need to do is send the ASCII number or identifier to NAME TABLE and the correct character will be displayed on the VDU (TV picture).

This process is a lot simplier and less CPU intensive than having to send the 8 bytes that make up the shape everytime we want to write to the screen,instead of the ONE byte ASCII number. However,as we will see in chapter 6.0,the latter is essential for plotting points and positioning text as needed in Desktop Publishing ,DTP.

Register 3:

This register is used to define where the colour table is located in VRAM. The base address for the colour Table is calculated as follows:

```
-----------------------------------
 COLBASE = Register 3 [0-255] * 64
-----------------------------------
```

This register is only applicable in both graphic modes. The TEXT colour register is set using REG 7 and will be discussed later. The colour base can be located anywhere in VRAM in GI mode or GII (text mode). Obviously the location of this table will be governed by the other defined tables. However,in Graphics II,high resolution mode,this table can only be located at #0000 or 8192. In this case the Register 3 value is either #03 or #FF

Register 4:

This register is used to locate the pattern generator table in VRAM. Only 3 bits [range 0-7] are necessary ,and since this table is 2k long,and therefore can only be located on 2k boundries. Therefore,in the 16k VRAM,there are 8x2k boundries,see Table 2-6.

```
-----------------------------------
 PATBASE = Register 4 [0-7] * 2048
-----------------------------------
```

Table 2-6: Pattern Generator Boundry Starting Addresses.

Reg 4	Reg 4 * 2k
0	0000 or #0000
1	2048 or #0800
2	4096 or #1000
3	6144 or #1800
4	8192 or #2000
5	10240 or #2800
6	12288 or #3000
7	14336 or #3800

However, as with Register 3,in High resolution GII mode only the two special cases exist: #03 and #FF. Usually,Register 3 is #FF (8192) and register 4 is #03 ,ie located at 0000.

Register 5:

Bit 7 is set to Zero,whilst the other 7 bits are used to define the starting address of the Sprite Attribute Table,or SAT. A fuller discussion of this and Registers 6&8 can be found in Chapter 7. The start of this Table is calculated as:

```
------------------------------------
 SATBASE = Register 5 [0-127] * 128
------------------------------------
```

Register 6:

This 3-bit number defines the boundry starting position of the Sprite Pattern Generator Table.This table is calculated:

```
-----------------------------------
 SPGBASE = Register 6 [0-7] * 2048
-----------------------------------
```

See Table 2-6 for actual VRAM starting locations.

Register 7:

This register will be discussed in more detail in chapter 3.0.The MSN holds the INK colour ,whereas the LSN holds the PAPER colour. The colours are selected from a palette of 16. (see Table 3-1) Register 7 doesn't relate to a VRAM address but to the TEXT screen colour and is calculated as:

```
------------------------------------------------------------
 REGISTER 7 = (INK Colour [0-15] * 16 ) + PAPER Colour [0-15]
------------------------------------------------------------
```

Register 8:

This register will be dealt with in chapter 7.0.

2.4.2 Initialising the VDP Registers

The subroutine,VDPREG8SET,initialises the VDP registers according to the Register values held in VDPDATA.

Listing 2-1: Initialising the VDP Write Registers.

```
100 CODE

INITVDPREG:LD HL,VDPDATA    ; THE REGISTER DATA BYTES.
           CALL VDPREG8SET  ; SET THEM.
           RET              ; EXIT.

VDPREG8SET:LD BC,#0800      ; THERE ARE 8 VDP WRITE REGISTERS
REGWRTVDP: LD A,(HL)        ; TO INITIALISE. HL POINTS TO THE
           OUT (#02),A      ; 8 VDP DATA BYTES. THESE ARE SENT
           LD A,C           ; REGISTER AT A TIME TO THE VDP.
           OR #80           ; BIT 7 TELLS THE VDP THAT THE
                            ; REGISTERS ARE TO BE ALTERED.
           OR #40           ; BIT 6 TELLS THE VDP THAT WE ARE
           OUT (#02),A      ; WRITING TO THE VDP. THIS IS DONE
           INC C            ; UNTIL ALL 8 WRITE REGISTERS HAVE
           INC HL           ; BEEN SET.
           DJNZ REGWRTVDP   ;
           RET              ;EXIT SUBROUTINE.

VDPDATA:   DB #02,#C2,#0F,#FF,#03,#7E,#07,#16 ;GII HIGH RES.


110 RETURN
```

Also,listing 2-2,shows you how to alter a VDP register value during a program,see subroutine TXTSCRCOL. Basically,all you need to do is send the data byte followed by the register number and thats all. Finally ,section 2-6 ,at end of this chapter contains a few example VRAM layouts for you to try.

Listing 2-2: This subroutine alters the TEXT colours as stored in VDP Register 7.

```
120 CODE

TXTSCRCOL: LD A,(COL)       ;COL=(INK*16)+PAPER
           OUT (#02),A      ;SEND THE COL TO THE VDP.
           LD A,7           ;VDP REGISTER 7 IS SELECTED.
           OR #40           ;WE ARE WRITING TO THE VDP.
           OR #80           ;& ITS A VDP REGISTER.
           OUT (#02),A      ;THE REG,REG NUM & WRITE SENT.
           RET              ;EXIT SUBROUTINE.


130 RETURN
```

A subroutine to calculate the combined colour byte is given in chapter 3.0.

2.5 Accessing VRAM

All addressing throughout VRAM is 14-bit and this allows access to 2^14 or 16384 bytes of Video Ram. Addressing VRAM thus requires a two byte transfer, but only 14 of the 16 bits available are necessary for the VRAM address with bits 14 & 15 left over. These two bits tell the VDP whether READ or WRITE has been selected (bit 14 or MSB bit 6) and whether the reading or writing is to or from VRAM or the VDP REGISTERS (bit 15 or MSB bit 7). Table 2-7,gives the truth table for the bit 14/15 options. Table 2-8 on the otherhand gives a comprehensive breakdown of CPU and VDP data transfers.

Table 2-7: The bit truth table for VRAM addressing and for VDP Register updating.

Result of bit setting:	**WV**	**RV**	**WR**	**RR**
VRAM or VDP bit 15	0	0	1	1
READ or WRITE bit 14	1	0	1	0

where WV = Write to VRAM ; RV = Read from VRAM

WR = Write to VDP Regs ; RR = Read from VDP Regs

Note that the VDP requires a delay of 8 microseconds between successive Reads or Writes. A suitable delay would be PUSH AF/POP AF. When addressing VRAM,both the MSB and LSB are sent from the CPU to the VDP via PORT B. PORT A on the otherhand is only used for actual data transfers,ie Writing Sprite data or Reading the screen for screen dumps etc.

VRAM is managed by an autoincrementing register. This register is perfect for sequential data transfers via PORT A. In Sequential data transferring to or from VRAM, all you have to do is initialise the starting address of VRAM,and from that point onwards,no other address transfer is necessary as the autoincrementing pointer does this for you. However,if the data transfers are not sequential,then the VRAM address is required prior to each new data transfer to or from VRAM.

Table 2-8: A summary of CPU <-> VDP communication.
Where D signifies a data bit.
and A signifies an address bit.

Operation	**Byte**	**Description**								**Port**
	No.	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**used**
Write to	1	D7	D6	D5	D4	D3	D2	D1	D0	B
VDP REGISTER	2	1	1	0	0	RS	(3	-	0)	B
Read from	1	D7	D6	D5	D4	D3	D2	D1	D0	B
VDP REGISTER	2	1	0	0	0	1	0	0	0	B
Write to	1	A7	A6	A5	A4	A3	A2	A1	A0	B
VRAM	2	0	1	A13		to			A8	B
	3	D7	D6	D5	D4	D3	D2	D1	D0	A
Read from	1	A7	A6	A5	A4	A3	A2	A1	A0	B
VRAM	2	0	0	A13		to			A8	B
	3	D7	D6	D5	D4	D3	D2	D1	D0	A

The above table demonstrates that Writing and Reading the VDP registers requires only 2 data transfer using PORT B and Writing or Reading VRAM requires 3 bytes. The first two bytes specify the VRAM address [A0 - A13] and byte three contains the byte to be written two [D0 - D7] or holds the byte which has just been read. The latter byte operations are carried out using PORT A .

2.6 VRAM Map Examples

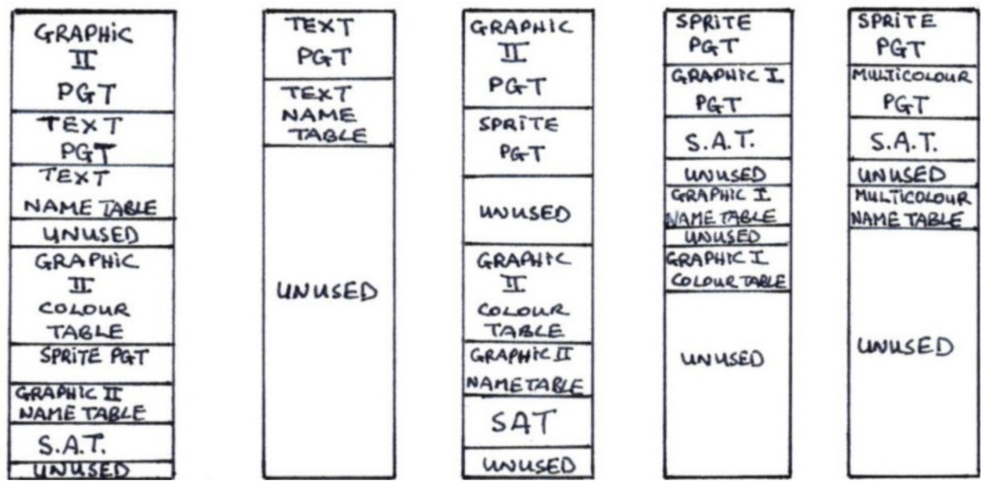
Table 2-9, summarises the VDP register values to set up the 5 VRAM mappings below. Notice how the IMG computer uses FOUR different VDP register setups, one for each of the IMG Screen types. The main drawback of this method,is that it destroys all the data held in VRAM, when Screen mode is changed. But it does mean that all four Screen modes are available in IMG BASIC. Whereas,the Memotech method is to set up VRAM so that the BASIC programmer can switch between TEXT and GII modes without losing the information on either,ie whats on the two screens is still intact. However,the MTX user cannot access Multicolour or GI screens from MTX BASIC only through a Z80 assembly program. The MTX method,tries to make use of all the available VRAM. The MTX switches between screens by altering VDP registers 0,1 and 2. Registers 0 and 1 select the screen mode and register 2, switches in the appropriate Name Table.

Table 2-9: The VDP register values for VRAM maps (a) to (e).

VDP Register Num	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**
(a) - GII	02	C2	0F	FF	03	7E	07	F4
(a) - TEXT	00	D2	07	FF	03	7E	07	F4
(b) - TEXT	00	D0	02	00	00	00	00	F5
(c) - GII	02	C2	0E	FF	03	76	03	0F
(d) - GI	00	C0	05	80	01	20	00	01
(e) - MULTICOL	00	CB	05	00	01	20	00	04

IMG = imaginary computer, in actual fact the MSX is set up like this, see MSX Technical Appendix.

(a) **MTX GII & TEXT** (b) **IMG TEXT** (c) **IMG GII** (d) **IMG GI** (e) **IMG MULTI**



3.0 TEXT Mode

3.1 Introduction

As the name suggests,only TEXT can be displayed in this mode. On the MTX and Einstein machines only the first 128 ASCII characters (0-127) can be displayed except for codes 0-31 as these are specially reserved non-printable control codes,refer the ASCII section of the Owners manual. On the otherhand,the MSX system has been designed to maximise the TEXT screen to the full by providing not only the 128 ASCII set but another 128 special MSX characters,providing MUSIC symbols,scientific characters,etc.

The ASCII character set is stored in the respective computers ROM and loaded into the Video RAM,VRAM at switch on. A character is designed typical as a 8x8 dot matrix, see Figure 3-1. A character requires typically to be 8-bits wide (1 byte) and needs to be 8 bytes deep.This gives the 8x8 character matrix. I find that this terminology is a little confusing as its 8 bits by 8 bytes. However,I will be discussing resolutions later which talk about dots. Here a dot is defined in pixels,ie 1x1,2x2,4x4. The same terminology is used on both sides on the multiplier sign.

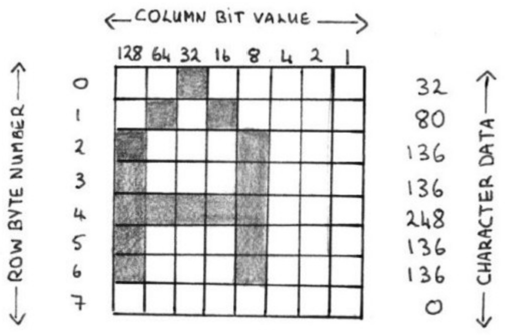


Figure 3-1: Character Designer Board (8x8 grid)

By using only 6 of the 256 screen wide pixels and 8 of the 192 screen height pixels to represent an ASCII character on the screen,the VDP can display a maximum of 40 (256/6) characters across the screen and 24 (192/8) characters down the screen for a 6x8 character matrix.This is ideal for Diary ,Notepad ,card indexing ,information retrieval and simple wordprocessing programs.

3.2 Character Compression/Expansion by Rotation

As shown in Figure 3-1,only 5 of the 8 bits in the screen width byte are necessary to represent the ASCII character,with bit 6 being left blank,so as to distinguish neighbouring characters on the screen. Also bits 7 & 8 are not used as in TEXT mode only 6 of the 8 bits are used. Forty (5x8) bits,ignoring the blank 6th bit, are needed to represent an ASCII character instead of 64 (8x8) bits. Therefore,we are wasting 24 bits or 3 bytes of RAM per character.

The Memotech ROM (page 0) at #35B3 stores the 96 printable ASCII characters (from 32 to 127). In order to reduce the above memory waste-age, the MTX programmers rotated the 8 bytes of screen data for the 6x8 character into 5 bytes (ignoring the blank bit) giving a 8x5 90o rotated form. Thus instead of 8*96 = 768 bytes to store the ASCII printable character set it only required 5*96 = 480 bytes. Listing 3-1 demonstrates how to rotate back to printable characters and Figure 3-3 explains how this is done. See also the screen dump chapter for a use of the rotated form of the ASCII character set.

Listing 3-1: Rotating the compressed 8x5 ROM character data
to the 6x8 screen printable character.

```
1000 CODE

           JR ROTSTART      ; bypass the variables.

VRAMASC:   DW #1900         ; where to store the ASCII in VRAM.
ROMASC:    DW #35B3         ; where the upturned ASCII in ROM.
PORTA:     DB #01           ; tells the VDP to write/read data.
PORTB:     DB #02           ; tells the VDP to select VRAM
                            ; address & to read or write data.


ROTSTART:  EX AF,AF'        ; save registers,AF,BC,DE,HL
           EXX              ; from corruption by this prog.
           LD HL,(VRAMASC)  ; where the ASCII char set is
                            ; going to be stored in VRAM.
           CALL VDPWRTSEL   ; This subroutine sets the VDP
                            ; to the correct VRAM address for
                            ; writing the ASCII set to.
           LD HL,(ROMASC)   ; the position in the MTX ROM of
                            ; upturned ASCII (8x5) char set.
           LD B,96          ; only 96 printable ASCII [32-127].
MAINLOOP:  PUSH BC          ; save this for testing later.
           LD B,8           ; Converting into 8 screen bytes
OUTERLOOP: LD C,5           ; From 5 ROM bytes.
           XOR A            ; clear register (stores screen ASC)
           PUSH HL          ; save current start of ROM ASC char
INNERLOOP: SLA A            ; make space for new bit at bit 0.
           LD E,(HL)        ; Get the upturned data byte pointed
                            ; to by HL.
           SLA E            ; move the upturned data byte left
                            ; by 1,catching the displaced bit 7
                            ; which will be stored carry flag.
           LD (HL),E        ; save the shifted upturned byte
           JR NC,NOTSET     ; if C=0 then shift 0 into reg A.
           SET 0,A          ; else place a 1 in reg A.
```

```
NOTSET:    INC HL           ; get the next byte in the sequence
                            ; of 5 to have a column stripped off
           DEC C            ; Each byte contibuting 1 bit to the
           JR NZ,INNERLOOP  ; screen ASCII character.
WRTVRAM:   OUT (#01),A      ; as each byte of the 5x8 char is
                            ; reformed, it is sent to VRAM via
                            : PORT A. NB: VRAM will move its
                            ; own automatic address pointer by 1
           POP HL           ; restore the ROM start address of
                            ; character being reformed to ASCII.
           DJNZ OUTERLOOP   ; get next row of char until all 8
                            ; screen bytes have been reformed.
           LD C,5           ; BC=0005,the displacement between
           ADD HL,BC        ; the upturned ROM chars.
           POP BC           ; decrease the printable ASCII
           DJNZ MAINLOOP    ; until all 96 have been reformed.
           EX AF,AF'        ; restore registers.
           EXX              ;
           RET              ; end of the prog.

VDPWRTSEL: PUSH AF          ; stops the flags from corruption
           LD A,L           ; A=LSB of VRAM address.
           OUT (#02),A      ; send to the VDP via PORTB.
           LD A,H           ; A=MSB of VRAM address.
           OR #40           ; select write data to VRAM.
           OUT (#02),A      ; send MSB to VDP via PORTB.
           POP AF           ; restore AF.
           RET              ; end of subroutine.


1010 RETURN
```

Save as:

```
     SAVE "ROTATETXT"                        (tape users)
     DISC (USER) SAVE "ROTATE.TXT"           (disc users)
```

As a check of the above algorithm,reload the above source text with LOAD "ROTATETXT" or DISC (USER) LOAD "ROTATE.TXT". Once installed,add the following lines to the above listing.

```
10 VS 4:CLS
20 GOSUB 1000
30 STOP
```

Before running the above,edit the code at 1000. VRAMASC should be changed to point to DW #0000. This change will also recompile the code to account for the BASIC text inserted at the start.

RUN <RET>

You will see the character set in the correct screen format,displayed on the graphics screen for demonstration. In reality,this character set will be hidden at 6144 of VRAM,the TEXT Pattern Generator Table.

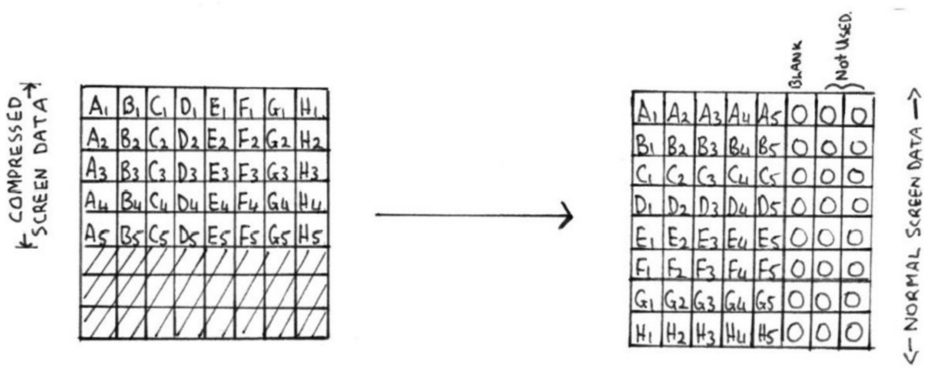


Figure 3-2: Rotation of the compressed 8x5 ROM character to a screen orientated 6x8 character.

3.3 Colour on the TEXT Screen

The TEXT screen can only have one background or PAPER colour and one foreground or INK colour,selected from any of the 16 possible colours, see Table 3-1. The non-active part of the screen or BORDER defaults to the PAPER colour. The VDP Register 7 holds the INK & PAPER colour of the TEXT screen. Figure 3-3,shows how the TEXT screen colour is represented. This technique of representing both the INK & PAPER colours in a single byte is also used in GI and GII modes. However,in the latter modes, 32 and 6144 bytes of colour information is required for these screens respectively.

Table 3-1: The 16 colours available for the TMS 9928/29 VDP.

COLOUR	**HEX value**	**Decimal Value**
Transparent	00	00
Black	01	01
Medium Green	02	02
Light Green	03	03
Dark Blue	04	04
Light Blue	05	05
Dark Red	06	06
Cyanate	07	07
Medium Red	08	08
Light Red	09	09
Dark Yellow	0A	10
Light Yellow	0B	11
Dark Green	0C	12
Magenta	0D	13
Grey	0E	14
White	0F	15

The upper 4 bits,the so called Most Significant Nybble,MSN, holds the INK (foreground) colour & the lower 4 bits,LSN,holds the PAPER (background) colour. The full byte is the colour attribute and is calculated as follows:

```
-------------------------------------------
 Colour byte value = ( 16 * INK ) + PAPER
-------------------------------------------
```

See GETCOL subroutine for the code to perform the above calculation. One byte holds the colour information for the entire TEXT screen.

```
INK   = #01
      =Black

Paper = #09
      = Light Red
```

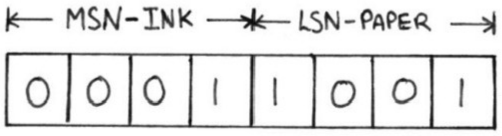


Figure 3-3: Colour Attribute Representation.

Listing 3-2 illustrates how we can set the text INK and PAPER colours from assembly language. This example keeps the INK colour constant but toggles the PAPER colour from 1 to 13. To select the PAPER Toggle press Function key <F1>.

Listing 3-2: PAPER toggle in TEXT mode.

```
1100 CODE

KEYS:       XOR A            ; reset Z-flag.
            CALL #0079       ; use MTX key scan routine to
            JR Z,KEYS        ; check for F1.
            CP 128           ; Is key=<F1>.
            CALL Z,PAPERTOG  ; toggle the Paper colour.
            CP 27            ; press <ESC> to end,ie
            RET Z            ; return to BASIC.
            JR KEYS          ;

PAPERTOG:   PUSH AF          ; save current keypress.
            LD A,(PAPER)     ; Get current PAPER colour.
            CP 14            ; If >13 then reset to 1.
            JR C,GT13        ; if <13 then increase PAPER
            XOR A            ; col by 1. Else reset it to
GT13:       INC A            ; one or increment by one.
            LD (PAPER),A     ; store new value.
            CALL GETCOL      ; update TEXT screen colours.
            CALL TXTSCRCOL   ; send it to VDP register 7.
            POP AF           ; restore keypress,so as not to
            RET              ; invoke another option in the
                             ; KEYS menu.
```

```
GETCOL:     PUSH BC          ; don't affect BC register.
            LD A,(INK)       ; this part loads the MSN with
            AND #0F          ; the INK colour.
            SLA A            ;
            SLA A            ; Equivalent to Multiplying
            SLA A            ; by 16 (ie 2^4).
            SLA A            ; INK in position in col byte.
            LD B,A           ; save it temporarily.
            LD A,(PAPER)     ; now load the PAPER into the
            AND #0F          ; LSN. NB:Mask unwanted bits.
            ADD A,B          ; COLour byte is formed.
            LD (COL),A       ; save it. NB:This subroutine
            POP BC           ; can be used as is in GI & GII
            RET              ; add to Library of Z80 code.

COL:        DS 1             ; stores the colour attribute.
INK:        DB #0F           ; defaults to White ink.
PAPER:      DB #01           ; defaults to black paper.

TXTSCRCOL:  LD A,(COL)       ; update the TEXT screen colour
            OUT (#02),A      ; via VDP register 7.
            LD A,7           ; select reg 7.
            OR #40           ; select WRITE to VDP.
            OR #80           ; select VDP reg to write too.
            OUT (#02),A      ; VDP reg 7 updated with new
            RET              ; colour attribute byte.

1110  RETURN
```

Save as:

```
            SAVE "TXTCOLOURTXT"               (tape users)
            DISC (USER) SAVE "TXTCOLOUR.TXT"  (disc users)
```

To test it simply enter the following:

```
10 VS 5:CLS
20 GOSUB 1100
30 STOP
```

Note the code at line 1100 started at #8007,but after the addition of lines 10,20 & 30,this code when recompiled moves to #8020. AS.1100 <RET> then <CLS> <RET> recompiles the code at 1100 to take into account the new BASIC text and points all the JP and CALL and system variables to the new RAM locations or addresses.

RUN <RET>

Every time you Press <F1> the PAPER colour will move to the next colour in the list,see Table 3-1. Only colours 1 to 13 are valid in this example. This cycle is in a continuous loop.

3.4 VRAM Tables

3.4.1 The TEXT Pattern Generator Table

As already stated, the ASCII character set (well the 96 printable ones) are stored from 6144 to [6144+(8*98)] in the VRAM map. As shown in Figure 3-1,each character requires 8 pattern bytes to make up its shape (note that the bottom byte is nearly always zero ,so that text from one row can be distinguished from the next row). In order to find the correct TEXT pattern for a particular character in VRAM,we use the following:

```
-------------------------------------------------------------
 TEXT PATTERN POSn = PATBASE + ( 8 * (ASCII Num - ASCII Sp ))
    in VRAM
-------------------------------------------------------------
```

Where:

ASCII num	=	a printable ASCII character between 32 & 127
ASCII Sp	=	the first printable ASCII character (number 32).
PATBASE	=	the start of the pattern table in VRAM,ie 6144.

For example,the capital letter 'A' pattern can be found at 6408 to 6415, ie VRAM start for pattern 'A' = 6144 + (8 * (65-32)) = 6408. Note that the ASCII number of the letter 'A' will always be 65 no matter the computer. The TEXT screen makes use of this.

3.4.2 TEXT Name Table

Whenever,the computer needs to display a character on the screen,all the Operating System,OS,has to do is send the TEXT Name Table VRAM address (on the MTX this is between 7168-8127) depending on the cursor position to the VDP. This is followed by the ASCII character number [ie between 32-127]. The VDP will automatically store this ASCII number at the VRAM address. The VDP will then extract the pattern corresponding to this ASCII number from the TEXT Pattern Generator Table. The VDP will then echo this to the TV picture or VDU.

We see the TEXT screen as a Rectangular area on the VDU,with a 40x24 Text screen resolution. However,in VRAM,the TEXT Name Table which keeps track of what is displayed on the VDU,is really a sequential block of 960 bytes of VRAM,see figure 3-4. Therefore,to relate the TEXT coordinate system to a VRAM address is a simple matter, see SETVRAM subroutine ,Listing 3-3 and the calculation which follows.

```
-------------------------------------------------------------
 NAME TABLE POSITION in VRAM = NAMEBASE + xpos + ( ypos * 40 )
-------------------------------------------------------------
```

where xpos = 0 to 39 and ypos = 0 to 23 and NAMEBASE=7168.
Note that the NAMEBASE is held in VDP register 2.

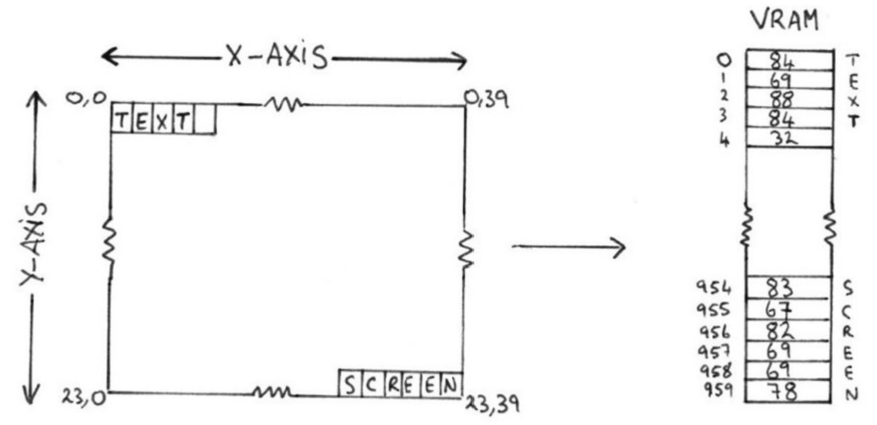


Figure 3-4: The TEXT screen and the TEXT Name Table.

The following listing will show you how to use the above to write text to the TEXT screen:

Listing 3-3: Writing to the TEXT screen. This example echoes what you type at the keyboard to the screen.

```
1200  CODE

;this routine assumes that the VDP's registers  have been
;setup for the TEXT screen to start at VRAM= 7168 and for
;ASCII character set to start at VRAM = 6144.  MTX  BASIC
;sets these values to the appropriate registers at switch
;on.

EXAMPLE:    LD HL,#0000       ; set xpos=0 and ypos=0,ie top LHC.
            LD (XY),HL        ; xpos & ypos stored.
                              ; Set the VDP auto incrementing
            CALL SETVRAM      ; address pointer to the TEXT SCRN.
GETCHAR:    XOR A             ; reset the Z-flag.
            CALL #0079        ; MTX ROM routine for getting a
            JR Z,GETCHAR      ; keypress (stored in Reg A). The
                              ; Z-flag is set if no key pressed.
            CP 27             ; if ESC is pressed then end.
            JR Z,ENDCODE      ;
            CP 32             ; if keypress is not in the ASCII
            JR C,GETCHAR      ; printable region of 32 to 127 then
            CP 128            ; try again,until valid.
            JR NC,GETCHAR     ;
VALIDCHAR:  OUT (#01),A       ; send ASCII number to PORT A.
            JR GETCHAR        ; there is no check for screen end.

ENDCODE:    RET               ; return to BASIC.


                              ; subroutine & system variables used.
```

```
SETVRAM:    PUSH AF          ; saved main registers. AF holds the
            EXX              ; ASCII keypress.
            LD HL,(TEXTSCRN) ; The start of the TEXT SCRN in VRAM
            PUSH HL          ; to be used latter on.
            LD BC,(XY)       ; B=ypos and C=xpos.
            XOR A            ; A=0
            LD HL,#0000      ;
            CP B             ; A=B=0 then row 0,don't add 40 or
            JR Z,ROW0        ; VRAM will be on the wrong row.
            LD DE,(SCRNWIDE) ; screen width.
LOOP1:      ADD HL,DE        ; this gives ypos*40
            DJNZ LOOP1       ;
ROW0:       POP DE           ; DE=7168
            ADD HL,DE        ; this gives 7168+(ypos*40).

            LD A,L           ; A=LSB of the current address
            ADD A,C          ; this gives:
            LD L,A           ; 7168+xpos+(ypos*40).
            CALL VDPWRTSEL   ;
            POP AF           ; restore keypress.
            EXX              ; restore other register pairs.
            RET              ; end of subroutine.

VDPWRTSEL:                   ; insert the code for this subroutine here
                             ; see listing 3-1 for the source text.

TEXTSCRN:   DW 7168          ; start of TEXT SCREEN in VRAM.
                             ; ie start of the Name Table.
XY      :   DS 2             ; holds the xpos & ypos values.
SCRNWIDE:   DW 0040          ; screen width.


1210 RETURN
```

Save as:

```
            SAVE "WRITETXT"                      (tape users)
            DISC (USER) SAVE "WRITETXT.TXT"      (disc users)
```

To test listing 3-3,add the following:

```
10 GOSUB 1200
20 STOP
```

As before recompile the code at 1200 and RUN <RET>.

Note that the above listing has no error checking for screen end. Also because the VDP has an autoincrementing VRAM address pointer,we only use SETVRAM subroutine once. This routine works like the MTX CSR x,y command for positioning text on the screen.The bulk of the above could be adapted as a CSR x,y command equivalent in Z80 assembly language. Notice that the above routine also uses VDPWRTSEL subroutine from the previous listing. It is possible to build up a library of such subroutines to make a programmers life easier.

4.0 **Graphics I and Multicolour Modes**

4.1 Graphics I Mode

4.1.1 Overview

Both the Memotech MTX and the Tatung Einstein machines do not support this mode from their respective BASIC Interpreters. The Japanese MSX system supports all 4 hardware screen modes from BASIC. Even though the Einstein and MTX machines BASIC Interpreter doesn't support it,you are able to reconfigure the VDP from the Z80 assembler.

Graphics mode I,is a more colourful! and characterful version of the TEXT screen. GI mode has a screen resolution of 256x192 pixels. However,it is a text orientated graphic screen because of its limited colour capabilities. It offers the user 256 User Defineable Graphics,UDG's. Each UDG is defined as an 8x8 matrix and therefore the screen text resolution is 32x24. The screen can only support 2 colours per block of 8 characters.

I may have been a bit unfair to GI mode when I implied it was a text orientated screen. In actual fact,it has the same resolution as the ZX Spectrum without the 2 colours per character but with the addition of 32 hardware sprites,see the chapter 7.0 . Also ,in GI , mode less VRAM is required than GII mode (see next chapter) and this extra Video Ram can be used for storing other UDG's or graphical information,ie like a second screen,without using up valuable CPU RAM as in other machines.

4.1.2 Fast Screen Switching

The advantage of being able to store a second screen or character set in VRAM is very useful,because,all that is needed to switch to this new screen,is to change the value of the VDP register which holds the starting address of the pattern table. Obviously this is a lot faster than having to block move all the new screen information to the screen,1 byte at a time. The example described in Listing 4-1,simulates a very fast screen change. However,as I am no artist,I have decided to demonstrate this principle by swapping the onscreen font by a user defined font. This example also introduces the user to VDP setup,this was discussed in chapter 2.0 .

Before we can precede with Listing 4-2,we will first have to define an alternative character set. I have listed below an alternative ASCII character set based on a 8x8 matrix. When you have typed in all the data,save it to tape or disc as a source file,so that errors can be corrected later.

```
SAVE "RAMASCIITXT"                          (tape users)
DISC (USER) SAVE "RAMASCII.TXT"             (disc users)
```

Listing 4-1: The alternative ASCII character set.

```
1300 CODE

    DB 0,0,0,0,0,0,0,0                      ; space     ( 32 )
    DB 16,16,16,16,16,0,16,0                ; !         ( 33 )
    DB 72,72,72,0,0,0,0,0                   ; "         ( 34 )
    DB 72,72,252,72,252,72,72,0             ; #         ( 35 )
    DB 16,124,144.124,20,124,16,0           ; $         ( 36 )
    DB 132,136,16,32,64,132,132,0           ; %         ( 37 )
    DB 112,136,112,96,148,136,112,0         ; &         ( 38 )
    DB 16,32,64,0,0,0,0,0                   ; '         ( 39 )
    DB 48,64,128,128,128,64,48,0            ; (         ( 40 )
    DB 48,8,4,4,4,8,48,0                    ; )         ( 41 )
    DB 132,72,48,252,48,72,132,0            ; *         ( 42 )
    DB 0,16,16,124,16,16,0,0                ; +         ( 43 )
    DB 0,0,0,0,48,16,32,0                   ; ,         ( 44 )
    DB 0,0,0,0,0,252,0,0                    ; -         ( 45 )
    DB 0,0,0,0,0,12,12,0                    ; .         ( 46 )
    DB 4,8,16,32,64,128,128,0               ; /         ( 47 )

    DB 48,72,140,148,164,72,48,0            ; 0         ( 48 )
    DB 48,80,16,16,16,16,124,0              ; 1         ( 49 )
    DB 56,68,4,8,16,32,124,0                ; 2         ( 50 )
    DB 120,132,4,60,4,132,120,0             ; 3         ( 51 )
    DB 128,128,128,144,144,252,16,0         ; 4         ( 52 )
    DB 248,128,128,248,4,4,248,0            ; 5         ( 53 )
    DB 48,64,128,248,132,132,120,0          ; 6         ( 54 )
    DB 252,4,8,16,32,64,128,0               ; 7         ( 55 )
    DB 120,132,132,120,132,132,120,0        ; 8         ( 56 )
    DB 120,132,132,120,16,32,64,0           ; 9         ( 57 )

    DB 0,0,16,16,0,16,16,0                  ; :         ( 58 )
    DB 0,0,16,0,48,16,32,0                  ; ;         ( 59 )
    DB 24,32,64,128,64,32,24,0              ; <         ( 60 )
    DB 0,0,252,0,0,252,0,0                  ; =         ( 61 )
    DB 96,16,8,4,8,16,96,0                  ; >         ( 62 )
    DB 48,72,4,8,48,0,32,0                  ; ?         ( 63 )
    DB 120,132,4,52,84,84,56,0              ; @         ( 64 )

    DB 48,72,132,132,252,132,132,0          ; A         ( 65 )
    DB 248,132,132,248,132,132,248,0        ; B         ( 66 )
    DB 120,132,128,128,128,132,120,0        ; C         ( 67 )
    DB 240,136,132,132,132,136,240,0        ; D         ( 68 )
    DB 252,128,128,248,128,128,252,0        ; E         ( 69 )
    DB 252,128,128,248,128,128,128,0        ; F         ( 70 )
    DB 120,132,128,156,132,132,124,0        ; G         ( 71 )
    DB 132,132,132,252,132,132,132,0        ; H         ( 72 )
    DB 124,16,16,16,16,16,124,0             ; I         ( 73 )
    DB 124,4,4,4,132,132,120,0              ; J         ( 74 )
    DB 132,136,144,224,144,136,132,0        ; K         ( 75 )
    DB 128,128,128,128,128,128,252,0        ; L         ( 76 )
    DB 104,84,84,68,68,68,68,0              ; M         ( 77 )
    DB 132,132,196,164,148,140,132,0        ; N         ( 78 )
```

```
      DB 120,132,132,132,132,132,120,0        ; O          ( 79 )
      DB 248,132,132,248,128,128,128,0        ; P          ( 80 )
      DB 112,136,136,136,168,152,116,0        ; Q          ( 81 )
      DB 248,132,132,248,160,144,136,0        ; R          ( 82 )
      DB 120,132,128,120,4,132,120,0          ; S          ( 83 )
      DB 124,16,16,16,16,16,16,0              ; T          ( 84 )
      DB 132,132,132,132,132,140,116,0        ; U          ( 85 )
      DB 132,132,132,132,132,72,48,0          ; V          ( 86 )
      DB 204,132,132,132,132,180,72,0         ; W          ( 87 )
      DB 132,72,48,48,48,72,132,0             ; X          ( 88 )
      DB 68,68,68,56,16,16,16,0               ; Y          ( 89 )
      DB 252,4,8,16,32,64,252,0               ; Z          ( 90 )

      DB 240,128,128,128,128,128,240,0        ; [          ( 91 )
      DB 128,64,32,16,8,4,4,0                 ; \          ( 92 )
      DB 60,4,4,4,4,4,60,0                    ; ]          ( 93 )
      DB 16,40,68,0,0,0,0,0                   ; ^          ( 94 )
      DB 0,0,0,0,0,0,252,0                    ; _          ( 95 )
      DB 32,16,8,0,0,0,0,0                    ; '          ( 96 )

      DB 0,0,120,4,124,132,124,0              ; a          ( 97 )
      DB 0,128,128,248,132,132,120,0          ; b          ( 98 )
      DB 0,0,112,136,128,136,112,0            ; c          ( 99 )
      DB 0,4,4,60,68,68,60,0                  ; d          ( 100 )
      DB 0,0,120,132,252,128,120,0            ; e          ( 101 )
      DB 48,72,64,240,64,64,64,0              ; f          ( 102 )
      DB 0,0,120,132,124,4,132,120            ; g          ( 103 )
      DB 128,128,128,248,132,132,132,0        ; h          ( 104 )
      DB 16,0,48,16,16,16,16,0                ; i          ( 105 )
      DB 4,0,12,4,4,68,56,0                   ; j          ( 106 )
      DB 128,128,136,144,224,144,136,0        ; k          ( 107 )
      DB 48,16,16,16,16,16,16,0               ; l          ( 108 )
      DB 0,0,88,164,164,164,164,0             ; m          ( 109 )
      DB 0,0,184,196,132,132,132,0            ; n          ( 110 )
      DB 0,0,120,132,132,132,120,0            ; o          ( 111 )
      DB 0,0,248,132,248,128,128,0            ; p          ( 112 )
      DB 0,0,120,136,120,8,12,0               ; q          ( 113 )
      DB 0,0,184,196,128,128,128,0            ; r          ( 114 )
      DB 0,0,60,64,56,4,120,0                 ; s          ( 115 )
      DB 0,32,252,32,32,36,24,0               ; t          ( 116 )
      DB 0,0,132,132,132,140,116,0            ; u          ( 117 )
      DB 0,0,132,132,132,72,48,0              ; v          ( 118 )
      DB 0,0,132,132,132,180,72,0             ; w          ( 119 )
      DB 0,0,132,72,48,72,132,0               ; x          ( 120 )
      DB 0,0,132,132,124,4,132,120            ; y          ( 121 )
      DB 0,0,124,8,16,32,124,0                ; z          ( 122 )


      DB 48,64,64,128,64,64,48,0              ; {          ( 123 )
      DB 16,16,16,0,16,16,16,0                ; |          ( 124 )
      DB 48,8,8,4,8,8,48,0                    ; }          ( 125 )
      DB 32,84,84,136,0,0,0,0,0               ; ~          ( 126 )
      DB 252,252,252,252,252,252,252,252      ; DEL        ( 127 )

      RET                                     ; END OF DATA.

1310 RETURN
```

Listing 4-2: Setting up the VDP registers for a GI screen and
             how to use the extra VRAM for switching screens.
             As I am no artist,I will refrain from composing 2
             screens and show the screen switching by swapping
             the ASCII character table.

Reload listing 4-1.

```
         LOAD "RAMASCIITXT"
      or DISC (USER) LOAD "RAMASCII.TXT".
```

Now add the following lines of text:

```
10 CODE

8007        LD HL,#0000           ; move the code in line 1300 to
            LD DE,#9000           ; #9000 IN RAM.
            LD BC,768             ; 96 chars of 8 bytes = 768.
            LDIR                  ; block move it.
            RET                   ; exit

20 GOSUB 1400
30 STOP


1300 CODE
      insert the ascii character set above

1310 RETURN


1400 CODE

START:      LD HL,REGGIMTX        ; SET MTX VRAM MAP AS A
            CALL VDPREGSET8       ; GRAPHICS I TEXT MODE.
            CALL ROTSTART         ; STORE MTX ROM ASCII SET
                                  ; AT #0000+(8*32).
            LD HL,(VRAMASC2)      ; STORE THE NEW ASCII SET
            CALL VDPWRTSEL        ; AT 2048+(8*32). ONLY THE
            LD C,96               ; 96 PRINTABLE ONE'S ARE
            LD HL,RAMASC          ; STORED IN THE PATTERN
LOOP:       LD B,8                ; TABLE AND THAT'S WHY THE
LOOP2:      LD A,(HL)             ; 32*8 DISPLACEMENT. NOW
            OUT (#01),A           ; THAT THE VRAM AUTO
            INC HL                ; INCREMENTER IS PRIMED.
            DJNZ LOOP2            ; COPY THE RAMASC SET TO THE
            DEC C                 ; VRAM BLOCK AT 2304,FOR EACH
            JR NZ,LOOP            ; 8 PATTERN BYTES FOR EACH OF
                                  ; THE 96 CHARACTERS.

SCRNCLS:    LD DE,#0000           ; SET SCRN TO 0,0 & THIS ALSO
            CALL SETSCRN          ; ACTS AS A SCREEN LOCATOR,ie
            LD BC,768             ; DE=00 TO 767 (32x24). AT
SCRCLS1:    LD A,32               ; STARTUP BOTH CHARACTER SETS
            OUT (#01),A           ; ARE COPIED TO THE SCREEN &
            DEC BC                ; NEEDED TO BLANKED FROM THE
```

```
           LD A,B                 ; GI SCRN OR NAME TABLE,BUT
           OR C                   ; NOT FROM THE PATTERN TABLE.
           JR NZ,SCRCLS1          ; ALL WILL BE CLEAR LATTER.

INITSCRN:  CALL SETSCRN           ; SET SCRN TO 0,0.NB:DE HOLDS
                                  ; THE X,Y VALUES.STILL AT 0,0

GETCHAR:   XOR A                  ; THE CODE THAT FOLLOWS IS AN
           CALL #0079             ; UPDATE OF LISTING 2-2.IT NOW
           JR Z,GETCHAR           ; INCLUDES A CHECK FOR SCREEN

           CP 128                 ; END BY LOOKING AT REG DE.
           CALL Z,SWITCHMODE      ; IT ALSO CHECKS FUNCTION KEY
           CP 27                  ; F1,TO SEE IF THE ASCII SET
           JR Z,ENDCODE           ; TOGGLE IS SELECTED.
           CP 32                  ;
           JR C,GETCHAR           ;
           CP 128                 ; ONLY 32 TO 127 ALLOWED AS
           JR NC,GETCHAR          ; VALID PRINTABLE CHARACTERS.

                                  ;
VALIDCHAR: LD HL,768              ; IS SCREEN END BEEN REACHED.
           SBC HL,DE              ; IF AT THE END DON'T UPDATE
           JR C,GETCHAR           ; SCREEN OR CURSOR. WAITS
           INC DE                 ; UNTIL ESC HAS BEEN PRESSED.
           OUT (#01),A            ;
           JR GETCHAR             ; STAYS IN LOOP UNTIL ESC.

ENDCODE:   RET                    ; RETURN TO BASIC.

ROTSTART:  EX AF,AF'              ; MERGE THIS SOURCE TEXT WITH
                                  ; THAT OF THE ROTATE SOURCE
                                  ; TEXT AND INSERT HERE. NB:
                                  ; VRAMASC IS NOW VRAMASC1 AND
           RET                    ; IT NOW POINTS TO #0000+256.


VDPWRTSEL: PUSH AF                ; SUBROUTINE AS IN LISTING
                                  ; 2-1.
           RET                    ; END SUBROUTINE.


VDPREGSET8:LD BC,#0800            ; THERE IS 8 VDP REGISTERS TO
REGWRTVDP: LD A,(HL)              ; SET [R0-R7]. THESE DEFINE THE
           OUT (#02),A            ; VRAM MAP. SEND THE DATABYTE
           LD A,C                 ; FIRST FOLLOWED BY THE VDP
           OR #C0                 ; REG NUMBER.BIT 7&6 HAS TO BE
           OUT (#02),A            ; SET TO INDICATE WRITING TO
           INC C                  ; VDP REG's INSTEAD OF VRAM.
           INC HL                 ; HL POINTS TO THE DATA BYTES
           DJNZ REGWRTVDP         ; WHICH DEFINES VRAM.SEE LAST
           RET                    ; SECTION OF THIS CHAPTER.

SETSCRN:   LD HL,(GISCRN)         ; POINT THE VDP TO THE GI/GII
           ADD HL,DE              ; NAME TABLE OR SCREEN.REG DE
           CALL VDPWRTSEL         ; HOLDS THE CURSOR X,Y VALUES
           RET                    ; NB:NEED THIS INFO WHEN SCRN
```

```
; NB:  SWITCHING  CAUSES THE  VDP  POINTER  TO  MOVE
; ANOTHER PART OF VRAM. SINCE WE HAVEN'T RESTORED IT
; THE TEXT WILL BE SENT TO THIS NEW VRAM  AREA. THIS
; IS WHY THE TEXT APPEARS INVISIBLE.  IN ACTUAL FACT
; WE ARE CORRUPTING ANOTHER VRAM TABLE.


SWITCHMODE:PUSH AF              ; THIS SUBROUTINE SWITCHES
           LD A,(FLAG)          ; BETWEEN THE TWO ASCII SETS
           XOR 1                ; USING REG VDP 4 AS A TOGGLE
           LD (FLAG),A          ; AS ONLY TWO SETS OR SCREENS
           OUT (#02),A          ; ARE USED ,IT'S A SIMPLE
           LD A,#04             ; TOGGLE OF BIT 0. WHEN IT IS
           OR #C0               ; '0' THEN VRAM POSn = 0*2048
           OUT (#02),A          ; '1' THEN VRAM POSn = 1*2048
           CALL SETSCRN         ; REMEMBER ON SCRN SWITCHING
           POP AF               ; THE NAME TABLE POSn IN VRAM
           RET                  ; IS LOST & NEEDS UPDATING.

VRAMASC1:  DW 0256              ; POSn OF STORED ROMASC IN VRAM
ROMASC:    DW #35B3             ; POSn OF ROM ASC CHARACTER SET
VRAMASC2:  DW 2304              ; POSn OF STORED RAMASC IN VRAM
RAMASC:    DW #9000             ; STORE AT #9000 IN RAM.
GISCRN:    DW 15360             ; START OF NAME TABLE OR SCRN.
FLAG:      DB #00               ; CAN ONLY BE 0 or 1.DEF=1.
REGG1MTX:  DB #00,#C2,#0F,#9F,#00,#7E,#07,#16

1410 RETURN
```

Now recompile the code at 1300. Then edit line 10 and change HL to the new starting address of line 1300. Now Save:

```
SAVE "GISWITCHTXT"                          (tape users)
DISC (USER) SAVE "GISWITCH.TXT"             (disc users)
```

To run: RUN <RET>

Obviously,switching fonts as in this example is useless,but the ability to switch screen displays as in a program like MANIC MINER will greatly speed up graphics and because of the extra VRAM space,it is possible to store upto 7 full screen displays if you configure VRAM correctly. This also releases valuable CPU RAM ,especially if you want your program to run on a 32k Ram machine.

4.1.3 Library Building

Example VRAM maps similar to the example above is at the end of chapter 2.0. The above program,involved little development time because the majority of the code was already available as re-usable subroutines,ie like procedures in PASCAL , MODULA ,etc. It is very important to spend longer on intial program development in order to build up a library of powerful MACRO's as they are called in assembly language. You will reap the benefits later on.

4.1.4 VRAM Tables - Pattern & Name Tables

Listing 3-3,can also be used on the Graphics I or II screens. I will leave the latter to you to solve. To give you a clue,you will have to glue bits of Listing 4-2 with it,ie the VDPREGSET8,etc. The formula to calculate the start of the Name Table and the position of a character shape in the Pattern table has already been covered in the TEXT section. NB: the NAMEBASE=15360 and PATBASE=0000.However,the calculation for the character in the pattern table is different than in the text mode as you do not need to account for the 32 control codes,as all 256 UDG's are printable. Thererfore the (-ASCII Sp) part is to be ignored.The values of NAMEBASE & PATBASE are dependent on the VRAM configuration. Also the ypos range in GI mode will be 0-31 instead of 0-39 for TEXT Mode.

Figure 4-1,below,shows how the VDP builds up a character on the screen which you view. The VDP's registers point to specific areas in VRAM. In GI mode the pattern table is stored at 0000, the colour table at 8192 and the Name Table at 15360. When the user presses a key,the ASCII number that represents this key,is then sent to the VDP.The VDP stores this number in the name table,ie the letter 'A' would be stored in the Name table as '65'. The VDP would then get the pattern for ASCII 65 from the pattern table,and from the colour table the appropriate colour and then echo this information to the VDU.

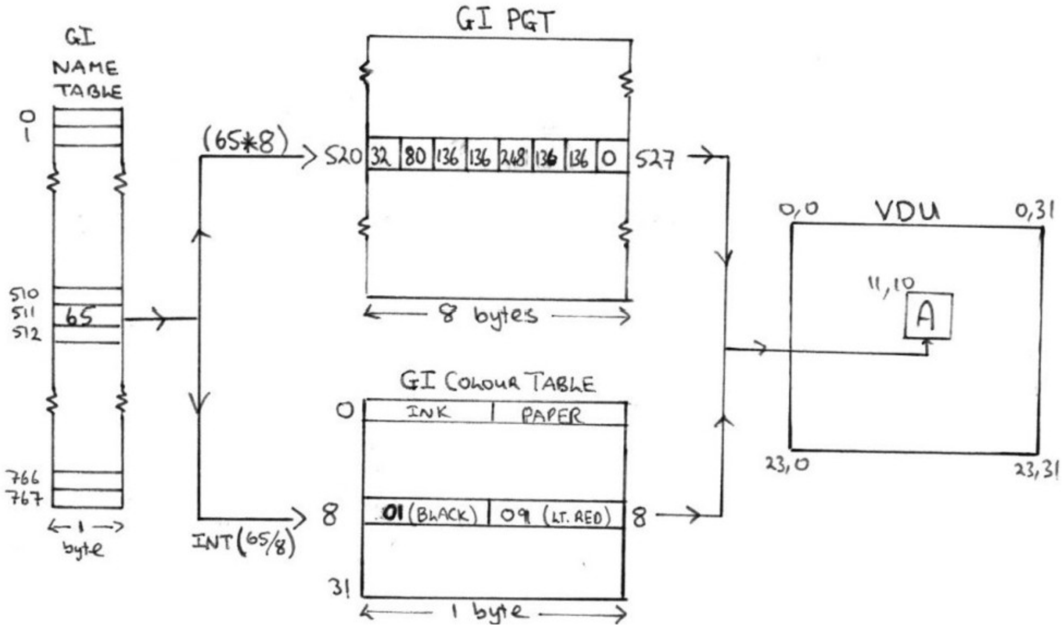


Figure 4-1: How the three VRAM tables (NAME ->PATTERN ->COLOUR) are used to make up a Graphics picture on the VDU in GI mode.

4.1.5 Colour

For every eight sequential ASCII patterns held in the PGT, there are two corresponding colours (one INK & one PAPER). This poor colour resolution - although better then in TEXT mode - is a serious disappointment. As shown in Figure 4-1, the GI colour table is only 32 bytes long [0-31]. With each location in the 32 byte table, holding a single colour byte ,see Figure 3-3 for the colour byte representation. In order to calculate which of the 32 colour locations is responsible for setting the colour of a particular pattern is calculated thus:

```
----------------------------------------------
 GICOL = GICOLBASE + INT ( ASCII Number / 8 )
----------------------------------------------
```

where GICOLBASE = start of the colour table in VRAM

For example,the colour location that holds the colour of the the letter 'A' (ASCII = 65) is GICOLBASE + 8.

Listing 4-3: Setting the colour of a ASCII character in GI mode or colouring a screen byte in GII mode using the subroutine below.

```
140 CODE


GSCRCOL:   LD A,(ASCIINUM)        ; CPU REG A holds the ASCII number.
           SRL A                  ;
           SRL A                  ;
           SRL A                  ; A = ASCIINUM/8
           AND A                  ; A = INT (ASCIINUM/8)
           LD HL,(GICOLBASE)      ; start of the colour table in VRAM.
           ADD A,L                ; get the colour position by adding
           LD L,A                 ; the colbase to the displacement.
           CALL VDPWRTSEL         ; tell VDP,see Listing 3-1.
                                  ; When GETCOL is called,it is
                                  ; assumed that INK & PAPER are set
                                  ; to the desired values.
           CALL GETCOL            ; see listing 3-2.Now get VRAM colour
           LD A,(COL)             ; byte to send to screen.
           OUT (1),A              ; send colour byte to screen.
           RET                    ; exit subroutine.


150 RETURN
```

Note that the relationship between GI colour mapping and GII colour mapping is completely different , see chapter 5.0 .

4.2 Multicolour Mode (64x 48 resolution)

This is the worst mode with respect to pixel resolution only 64x48. In this mode each dot is equivalent to 4x4 pixels in GI or GII mode. In actual fact,this mode doesn't deal with graphical pixels but with colour pixels. Each pixel can have its own unique colour,ie offers pixel colour but at such low resolution that its virtually non-descript.

Its okay for large chunky graphics,as would be used in kiddies early learning programs and for that reason I have mentioned it in this book. However,I will not discuss it but for more detail,I refer you to the comprehensive TI 9929A VDP technical manual,which should be easy to understand considering what you've hopefully learnt in this book.

5.0 **Graphics II Text Mode (32 x 24 characters)**

5.1 Introduction

The screen resolution of this screen is as for Graphics I mode , 256x192. Also,as in GI mode,three VRAM tables are required to generate a display on the Visual Display Unit,VDU. The tables are the Pattern Generator Table (where the shapes are stored) ;The Colour Table and the Name Table.

Graphics II mode can operate in one or two modes depending on the application - bit-mapped as required for plotting points and drawing lines as used in Desktop publishing,DTP, and Computer Aid Design,CAD ; or as a colourful Text screen as would be used for wordprocessors and front-end systems. The bit-mapped mode will be dealt with in chapter 6.0.

Both GI and GII modes have a name table of 768 bytes, giving a text resolution of 32x24. However, GII modes allows 768 UDG's as opposed to the normal 256 patterns. This allows a unique pattern to be created for every possible Name Table position. If this wasn't all,you can have 2 colours per byte or 16 colours per character (8 INK and 8 PAPER colours per 8x8 character).

5.2 Enhanced Resolution by Partition

This enhanced graphic and colour resolution requires about three quarters of VRAM. A screen resolution of 256x192 requires 49,152 bits (dots) or 6144 bytes of graphic information. The same number of bytes is needed for the colour table. Both the pattern generator and colour tables are segmented into 3 blocks of 2048 bytes. This block is further divided into 256 by 8 pattern or colour bytes (see figure 4-1,where the PGT of 2048 bytes in GI mode is equivalent to only one third of the GII mode PGT) .

Having a pattern and colour table of the same length means that finding the corresponding colour information for a particular character is a simple task.This is because,the formula for both pieces of information are identical except for the start of the respective VRAM tables.

```
-------------------------------------------------------------
Start of character = PATBASE + BLOCK + ( 8 * ASCII  Num )
in the PGT
-------------------------------------------------------------
Start of colour for = COLBASE + BLOCK + ( 8 * ASCII  Num )
char in the COLTAB
-------------------------------------------------------------
```

where:
```
BLOCK    = 0000 (top) or 2048 (middle) or 4096 (bottom)
COLBASE  = 8192 for a MTX
```

```
      PATBASE  = 0000 for a MTX
      ASCII Num= 0 - 255.
```

5.3 The VRAM Tables

As shown in figure 4-1, the name table is a sequence of 768 locations which hold a particular ASCII code [0-255]. As you are well aware,only numbers between 0 & 255 are allowable. This raises the question of how we access the other 512 UDG's. As already stated,the pattern generator and colour tables are divided horizontally into thirds of 2048 bytes . Well this is also the case for the Name Table,except that its in blocks of 256 bytes,see Table 5-1 . The ASCII Numbers 0-255 apply to all three blocks. However,depending on which third of the screen, the character is to be displayed in, determines which block of the PGT and Colour Table the information should be extracted from.

Table 5-1: Sectioning in GII mode increases resolution.

Thirds	**Name Table**	**Colour Table**	**Pattern Table**
TOP	000 - 255	0000 - 2047	0000 - 2047
MIDDLE	256 - 511	2048 - 4095	2048 - 4095
BOTTOM	512 - 767	4096 - 6143	4096 - 6143

Please note that the actual positions of all three tables depends on the VRAM setup. The values quoted above are the undisplaced ranges. VDP registers 2,3 and 4 hold the base addresses of the Name Table,Colour Table and Pattern Tables in VRAM respectively. What you then do is add to the above displacements the Base values,ie 15360 (NAMEBASE),8192 (COLBASE) and 0000 (PATBASE) respectively for the MTX.

As with GI mode the user only needs to send the appropriate ASCII code and depending on the cursor position on the screen ,the correct pattern from the correct 2048 byte block will be selected automatically by the VDP,see later for matching Name Table positions with the TEXT coordinates.

5.4 Initialisation of the PGT

Problems will arise if the patterns in the three thirds of the PGT are not mapped the same . The example below will highlight this problem.

Listing 5-1 is simply a reworking of listing 4-2. There is no toggling between two character sets on the whole screen. This time ,as you move from the TOP to the MIDDLE third of the screen,you will notice that the font from this point will be differnt from the font above. But, when you move from the MIDDLE to the BOTTOM third of the screen, you will see nothing appearing on the screen. This is because no character set has been loaded into the bottom third of the PGT.

Listing 5-1: GII mode and the Pattern table.

Reload the source text of listing 4-2 with :

```
      SAVE "GISWITCHTXT"                      (for tape users)
      DISC (USER) SAVE "GISWITCH.TXT"         (for disc users)
```

Now Edit it with AS.1400 <RET>. The following changes and deletions are needed. Please do in the order set down. The CHANGES:

```
REGGIIMODE:DB #02,#C2,#0F,#FF,#03,#7E,#07,#16
GIISCRN:   DW 15360
SETSCRN:   LD HL,(GIISCRN)
START:     LD HL,REGGIIMTX
```

THE FOLLOWING LINES ARE REDUNDANT,THEREFORE DELETE:

```
FLAG:      DB #00

SWITCHMODE:PUSH AF to RET        ; ie the SWITCHMODE subroutine.

           CP 128
           CALL Z,SWITCHMODE
```

Now save the source text:

```
SAVE "GIISWITTXT"                   (for tape users)
DISC (USER) SAVE "GIISWIT.TXT"      (for disc users)
```

then RUN <RET>. Now type a couple of paragraphs of a book onto the screen. After 256 characters have been displayed the on screen font will change to the RAMASC font. NB:the previous 256 characters will stay in the old font mode.

When the MTX switches to VS 4 screen mode, the MTXOS initialises GII screen mode, by copying the ASCII characters into all three sections of the screen pattern table,similar to Listing 5-1,except that the font will not vary from section to section. Listing 5-2,describes how to initialise GII mode as for VS 4 mode in MTX BASIC.

Listing 5-2: Initialise all 3 sections of the Graphics II mode pattern table with the User Defined ASCII character set,in Listing 4-1. This example only loads the 96 printable ones. However,as this is a graphics orientated text screen, you could add your own patterns for ASCII characters >127,ie like Greek letters, italics , music symbols etc. Remember to change the number of ASCII characters to send to the pattern tables.

```
10 GOSUB 1500
20 STOP


1300 CODE

RAMASC:                         ; see listing 4-1.

1310 RETURN


1500 CODE

START:     LD HL,REGGIIMODE       ; SET VDP TO VS 4 MODE.
           CALL VDPWRTSEL         ;
           CALL INITASCTAB        ; LOAD 3 SECTIONS WITH ASC SET.
           CALL TXTSCRCLS         ; CLS THE NAME TABLE OF THE 3
           LD HL,(GIISCRN)        ; ASC SETS. THEN POINT THE CSR
           CALL VDPWRTSEL         ; TO THE START OF THE GII SCRN.
GETCHAR:   XOR A                  ;
           CALL #0079             ; READ KEYBOARD.
           JR Z,GETCHAR           ;
           CP 27                  ; RETURN TO BASIC WHEN "ESC" IS
           RET Z                  ; PRESSED.
           CP 32                  ; ONLY CHARS BETWEEN 32 AND 127
           JR C,GETCHAR           ; ARE VALID KEYPRESSES.
           CP 128                 ;
           JR NC,GETCHAR          ;
VALIDCHAR: OUT (#01),A            ; SEND ASCII CHAR NUMBER TO THE
           JR GETCHAR             ; NAME TABLE (SCRN).

INITASCPAT:LD HL,256              ; POSITION IN VRAM,SECTION 1,THAT
           LD DE,2048             ; THE PRINTABLE ASC CHARS ARE TO
           LD B,3                 ; STORED. DE=DISPLACEMENT TO NEXT
LDPATTAB1: CALL VDPWRTSEL         ; SECTION. THEY ARE 3 SECTIONS.
           PUSH BC                ; KEEP TRACK OF WHICH SECTION.
           PUSH HL                ; STORE ASC PRINTABLE START DISP.
           LD C,96                ; 96 PRINTABLE CHARACTERS TO BE
           LD HL,RAMASC           ; SENT TO EACH SECTION.RAMASC IS
                                  ; OBTAINED BY LOOKING AT LINE 1300
                                  ; & ENTERING ITS STARTING VALUE.
LDPATTAB2: LD B,8                 ; 8 BYTES PER CHARACTER ARE TO BE
LDPATTAB3: LD A,(HL)              ; START COPYING THE PATTERN BYTES
           OUT (#01),A            ; LISTED IN LISTING 4-1,INTO THE
           INC HL                 ; THREE SECTIONS OF THE PATTERN
           DJNZ LDPATTAB3         ; TABLE.
           DEC C                  ; UNTIL ALL 96 HAVE BEEN STORED.
           JR NZ,LDPATTAB2       ;
           POP HL                ; RESTORE PRINTABLE DISPLACEMENT
           ADC HL,DE             ; GET PRINTABLE START IN THE
                                 ; NEXT PATTERN SECTION.
           POP BC                ; REDUCE THE PATTERN SECTION
           DJNZ LDPATTAB1        ; BY 1,UNTIL ALL SECTIONS HAVE
           RET                   ; LOADED.

TXTSCRCLS: LD HL,(GIISCRN)       ; THIS SHORT SUBROUTINE CLEARS
           CALL VDPWRTSEL        ; THE GII NAME TABLE OR SCREEN
           LD BC,768             ; .THIS IS IMPORTANT,AFTER THE
TXTSCRCLS1:LD A,32               ; PATTERN IS LOADED,AS THE
           OUT (#01),A           ; PATTERN IS ECHOED TO THE NAME
           DEC BC                ; TABLE OR SCRN AND TO THE VDU.
           LD A,B                ; THIS SUBROUTINE HAS BEEN
           OR C                  ; MODULARISED SINCE LISTING 4-2
           JR NZ,TXTSCRCLS1      ; WHEN IT WAS CALLED SCRNCLS.
           RET                   ;

VDPWRTSEL: see listing 3-1       ;

VDPREGSET8:see listing 4-2       ;

GIISCRN:   DW 15360              ; NAME TABLE OR SCRN START IN VRAM.

REGGIIMODE:DB #02,#C2,#0F,#FF,#03,#7E,#07,#16;
```

Now save this:

```
           SAVE "INITVS4TXT"                (tape users)
           DISC (USER) SAVE "INITVS4.TXT"   (disc users)
```

As for Listing 5-1,type RUN <RET>,and start typing. Notice that the same font is used throughout the whole screen. I hope by now that you will have realised the potential of modular programming and building applications from these modules or building blocks. As you become more proficient with Assembly language,your library will start to swell and programs will be designed and debugged a lot quicker.

5.5 Colour Mapping with GII text screens

Figure 5-1,see later,graphically describes how the VRAM tables : Pattern ,Colour and Name relate to each other in GII text mode. As I will demonstrate by way of an appropriate example, see listing 5-3,that the way the GII TEXT mode is setup,you will not be able to use the colour table effectively because it behaves like the pattern generator table.

As already stated,both the pattern and colour tables are mapped similarly,ie same length in VRAM and each pattern byte has a corresponding colour byte,see the formula in section 5.2 . At present , we send the ASCII character number to the

NAME Table where this number is stored according to the current cursor coordinates. The VDP then finds out which third of the screen the character is to be displayed and then looks up the appropriate character pattern from the correct pattern table third.

The same principle is adopted in the way the VDP gets the colour of a particular character by reading the same location as in the PGT except that it is a further 8192 bytes higher up in VRAM. The significance of this is that once you have ascribed a particular colour pattern to a particular graphically pattern,this colour will stay with this pattern throughout that third of the screen. This means that it is impossible in this particular TEXT mode to change the colour of a specific character without affecting the previous colour of this character at different screen coordinates in the same third of the screen. Obviously this will cause problems.

Listing 5-3: This listing demonstrates the colour resolution problem discussed in the last few paragraphs.

```
1300 CODE
                                 ; SEE LISTING 4-1

1310 RETURN

1600 CODE

START:     LD HL,REGGIIMTX       ; SET  TO   GRAPHICS   II   MODE.
           CALL VDPWRTSEL        ;
           CALL INITASCTAB       ; INITIALISE THE PATTERN TABLE
                                 ; WITH THE RAM ASCII CHAR SET.
           LD A,31               ; INK=BLACK  AND PAPER=WHITE.
           LD HL,8712            ; LOAD THE COLOUR 8 BYTES WHICH
           CALL VDPWRTSEL        ; CORRESPOND TO THE PATTERN OF
           LD B,8                ; ASCII CHARACTER 'A",ie 65*8
COLLOOP1:  OUT (#01),A           ; VRAM=8192+520. ONLY COLOURING
           DJNZ COLLOOP1         ; THE 8 BYTES OF CHAR 'A' IN THE
                                 ; TOP THIRD OF THE SCREEN.
           CALL TXTSCRCLS        ; CLS THE SCREEN BEFORE USE.
           LD HL,(GIISCRN)       ; SET CURSOR TO 0,0.
           CALL VDPWRTSEL        ;
           CALL GETCHAR          ; ECHOES VALID ASCII CHARS
           RET                   ; [32-127] TO THE GII SCREEN.

; SUBROUTINES

INITASCTAB:                      ; SEE LISTING 5-2.

TXTSCRCLS:                       ; SEE LISTING 5-2.

VDPWRTSEL:                       ; SEE LISTING 3-1.

VDPREGSET8:                      ; SEE LISTING 4-2.
```

```
GETCHAR:    XOR A                  ;
            CALL #0079             ;
            CP 27                  ; ESC THEN EXIT
            RET Z                  ;
            CP 128                 ; INVERT COLOUR OF CHARACTER 'A'.
            CALL CHANGEACOL        ;
            CP 32                  ; MAKE SURE ONLY VALID CHARS ARE
            JR C,GETCHAR           ; SENT TO THE SCREEN.
            CP 127                 ;
            JR NC,GETCHAR          ;
VALIDCHAR:  OUT (#01),A            ;
            JR GETCHAR             ; ESC TO EXIT LOOP.

CHANGEACOL:PUSH AF                 ; DON'T CORRUPT ANY REGISTERS.
            PUSH BC                ;
            PUSH HL                ;
            LD HL,8712             ; CHANGE THE COLOUR OF LETTER 'A'
            CALL VDPWRTSEL         ; BY INVERTING IT.
            LD A,225               ;
            LD B,8                 ;
COLLOOP2:   OUT (#01),A            ;
            DJNZ COLLOOP2          ;
            POP HL                 ;
            POP BC                 ;
            POP AF                 ;
            RET                    ;


;SYSTEM VARIABLES


REGGIIMTX:  DB #02,#C2,#0F,#FF,#03,#7E,#07,#16
GIISCRN:    DW 15360
SCRNLEN:    DW 768

1610 RETURN
```

Save as:

```
            SAVE "G2TXTCOLTXT"                    (tape users)
            DISC (USER) SAVE "G2TXTCOL.TXT"       (disc users)
```

To test the above listing, simply enter the following:

```
10 GOSUB 1600
20 STOP
```

Remember to recompile lines 1300 and 1600 and update RAMASC in line 1600 to the start of line 1300. RUN <RET>. Now start typing text. Everytime you type the letter 'A' it will be in a white box with black ink. When you press <F1>,this will invert the colour of letter 'A' to white on black paper for all letter A's on the top third of the screen.

Also, after you switched the colour, anything else you type will not appear on the screen. We have already come across this problem. When you jump around VRAM, we are also changing the VRAM address pointer,in this case to 8712-8719. To regain control of it to the name table will require a special program pointer to reset the address pointer after a quick jump in the VRAM table,see listing 4-2,SETSCRN.

5.6 The Name Table and the TEXT Coordinate System

Relating the the TEXT screen coordinates as used by the MTX BASIC command CSR X,Y to positions in the Name Table is a simple task:

```
---------------------------------------------
 NAMETABPOS = NAMEBASE + ( Y * 32 ) + X
---------------------------------------------

where Y           = y-coordinate ,range 0-23.
      X           = x-coordinate ,range 0-31.
      NAMEBASE    = set according to VDP Register 2.
```

5.7 VDP Picture Mechanism/Screen Refresh

Without repeating myself too much,the VDP requires information from three specialised VRAM tables - PATTERN, COLOUR, and NAME, so that a computer generated TEXT orientated GRAPHICS screen can be produced on the VDU. Figure 5-1,graphically describes what information these tables hold and how they interact to produce the characters we see on the VDU. As you will have realised by now,that this mechanism is true for all TEXT orientated modes,except that as the resolution increases then the memory required to hold the information increases,ie more VRAM is used up.

In summary,the VDP sequentially reads the Name Table locations ,(NAMEBASE + 0) to (NAMEBASE + 767),converting the ASCII Numbers held in these locations into the desired shape with corresponding colour. This information is then sent to the VDU for display on the VDU picture display. This mechanism is repeated every 1/50 th of a second (depends on the Hz frequency - UK = 50 and US =60).

An illustration of the screen refresh procedure : If you change the pattern of ASCII 65 from 'A' to 'A',then the new pattern will almost instantaneously be echoed to the VDU picture. Not only this,but ALL other 'A's in the same third of the screen will be underlined simultaneously,because they have the same pattern. Obviously, this is a major drawback of this screen mode. However, the character pattern and colour uniqueness at different positions on the screen will be covered in chapter 6.0.

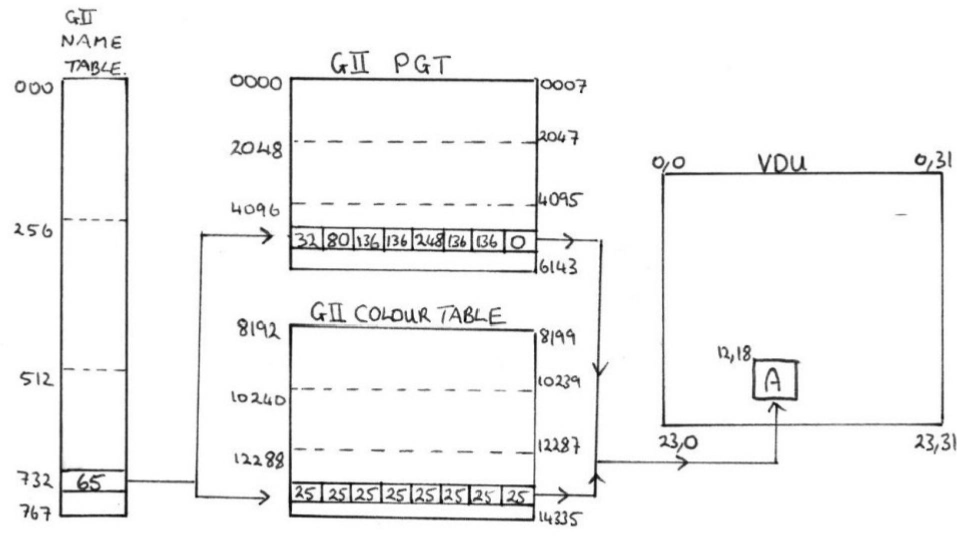


Figure 5-1: Interaction of the VRAM tables to give a character on the VDU picture.

5.8 GI Mode Emulation but with Enhanced Colour

Finally,as I said in the introduction of this chapter,that GII text mode was ideally suited to wordprocessing, spreadsheeting and for front-end systems. However,GII mode requires over 12k of VRAM,whereas in GI mode only 2-3k is needed. Because of the reduced VRAM constraints of GI mode,this mode can be used to store other screens,or alternative character sets,and as shown in chapter 4,it is possible to quickly switch fonts or entire screens.

Well,it is possible to emulate GI mode from within GII mode. This is easily done by setting Registers 0,3 & 4. The allowable values and there VRAM addresses is given in Table 5-2. It is obvious from this table,that only 3 other fonts or screens can be stored in VRAM,and likewise for the colour tables. At this point it is worth mentioning,that GI emulation,retains the colour prowess of GII mode. Therefore,for the 256 unique ASCII patterns,there are 256 equivalent colour patterns,unlike the 8 ASCII characters per colour pattern in normal GI mode.

Table 5-2: GI Emulation VDP Register Values & VRAM addresses. Includes GI values for comparison.

R0	**R1**	**R3**	**R4**	**COLBASE**	**PATBASE**	**SCREEN/FONT**
#02	#C2	#80	#00	8192	0000	one
#02	#C2	#A0	#01	10240	2048	two
#02	#C2	#C0	#02	12288	4096	three
#00	#C2	#80	#00	8192	0000	actual GI

Note that,all the information enclosed in chapter 4,is now pertinent to this section. Try a few of the listing with the above R0,R3 & R4 values. Once,you have convinced yourself of the emulation,type in the following short utility,to see the increased colour potential - ie 16 colours per unique ASCII character.

Listing 5-4: Illustration of the Increased Colour Resolution of GI & GII modes & GII mode emulating GI mode.

```
1700 CODE

COLRES:    LD HL,VDPREGGII         ; SET VDP REGISTERS  (*)
           CALL VDPREGSET8         ;
           LD HL,8192              ; THE START OF THE COLOUR TABLE.
           CALL VDPWRTSEL          ;
           XOR A                   ; SET COLOUR TO TRANSPARENT.
           EX AF,AF'               ;
           XOR A                   ; AF'=TRANSPARENT.
           LD BC,(BCGIIMODE)       ; LENGTH OF COLOUR TABLE.  (*)
COLLOP:    EX AF,AF'               ; GET START OF COL. RANGE=0-15.
           OUT (#01),A             ; SEND COLOUR BYTE TO COL TABLE
           INC A                   ; GET NEXT COL,SEE TABLE 3-1.
           CP 16                   ; IF COL=16 THEN RESET COL=0.
           JR NZ,NORESET           ;

RESETCOL:  XOR A                   ;
NORESET:   DEC BC                  ; DO THIS UNTIL THE WHOLE COLOUR
           EX AF,AF'               ; TABLE IS FILLED.SAVE THE UPDATED
           LD A,B                  ; COLOUR NUMBER IN AF'.
           OR C                    ;
           JR NZ,COLLOP            ;
           RET                     ;

; SUBROUTINES

VDPWRTSEL:                         ; SEE LISTING 3-1

VDPREG8SET:                        ; SEE LISTING 4-2

VDPREGGII: DB #02,#C2,#0F,#FF,#03,#7E,#07,#16      ; GII MODE.

VDPREGEMGI:DB #02,#C2,#0F,#80,#00,#7E,#07,#16      ; GII ->GI.

VDPREGGI:  DB #00,#C2,#0F,#80,#00,#7E,#07,#16      ; GI MODE.

BCGIIMODE: DW #1800    ;BC=6144    ; GII COLOUR TABLE SIZE.
BCGIIEMGI: DW #0800    ;BC=2048    ; GII EMULATING GI BUT WITH
                                   ; INCREASED COLOUR RESOLUTION
                                   ; EQUIVALENT TO A 3rd OF GII.
BCGIMODE:  DW #0020    ;BC=  32    ; GI COLOUR TABLE SIZE


1710 RETURN
```

Save as:

```
           SAVE "COLRESTXT"                        (tape users)
           DISC (or USER) SAVE "COLRES.TXT"        (disc users)
```

Add the following lines of BASIC to test the above:

```
10 VS 4:CLS
20 GOSUB 1700
30 GOTO 30
```

Now: RUN <RET>

When this program is run with the first set of data ,ie the VDP is configured as a high resolution GII mode,the you will see a multicoloured screen with every byte (8x1) a different colour. Every 16th byte,the colour will reset back to transparent and repeat colouring the succesive bytes according to table 3-1.Press <BRK> key to exit to BASIC.

Now,if we change the second of the two lines marked with the asterixes (*),to LD BC,(BCGIIEMGI),and leaving the Registers setup as normal GII mode,you will see only the top third coloured in. This is as expected,because we've only coloured 2048 bytes out of the 6144 bytes.

Next,change both *-lines to:

```
                      LD HL,VDPREGEMGI
                      LD BC,(BCGIIEMGI)
```

The Registers have been configured to emulate GI mode. When the program is rerun with this data,you will get the same effect as the first run ,even though we have only sent 2048 bytes of colour information. This is because we have reduced the size of the GII colour table from 6144 bytes to 2048. In this emulation,each of the 256 ASCII patterns has a unique 8x8 colour pattern,unlike the original GI mode,see below.

Finally,change the *-lines to:

```
                      LD HL,VDPREGGI
                      LD BC,(BCGIMODE)
```

When this is rerun with the normal GI register setup,you get the same colour for 8 8x8 patterns rather than 1 colour per 8x1 pattern in GII mode and in GII emulating GI mode.

The restrictions discussed in section 5.5 and demonstrated in listing 5-3,will hold for the Emulation mode also.

6.0 **Graphics II Bit Mapped Mode (256x192 dots)**

6.1 Introduction

Bit mapped screen display allows the programmer the flexibility to address (access) every dot on the GRAPHICS plane or screen. This access is essential for plotting points ,as shown in Listing 6-3,and drawing lines as required for paint/sprite designers,CAD,DTP,etc. The VDP has a screen resolution of 256x192 dots or the ability to plot 49k of dots or bits of graphical information. However,the VDP can only resolve 2 colours per byte of graphical information - see colour section for further explanation. In summary,the dot resolution is bit mapped but the colour resolution is byte mapped.

We have seen in this and in previous chapters,that the TEXT modes are interesting and useable to some extent,but they lack the flexibility and colourfulness of a bit mapped display.However,a bit mapped display will cost us in terms of

(1) VRAM - requires >12k.

(2) CPU RAM - required for colour and graphical data.

(3) Performance - CPU is needed to transfer a lot more information ,ie pattern and colour data. This makes the VDP less independent.

6.2 The Bit Mapped Mechanism wrt Text Orientated Displays

Both the TEXT mode and the BIT mapped mode share the same display mechanism as outlined in section 5.7. The TEXT mode,maps the 768 byte Name Table as a 32x24 VDU screen,with each of the Name Table Positions,NTP,corresponding to a CSR X,Y Text coordinate. When we want to display an ASCII character on the VDU,we would send its ASCII number to the correct NTP,where it would be stored. This ASCII number would correspond to one 8x8 shape in the PGT and one 8x2 colour in the Colour Table. The VDP would extract this information and echo it to the VDU.

The problem with this technique was that,if we wanted to change the colour (see listing 5-3) or the shape of a ASCII character then all ASCII characters with that number in that particular third of the screen or PGT had to adopt the new shape or colour according to the mechanism outlined in section 5.7. It for is this reason that you couldn't write a WYSIWYG wordprocessor in this mode.

In Bit mapped mode,we load the Name Table Positions with predefined numbers [0-255],see figure 6-1. This table like the PGT in the above mechanism is held in this state throughout switch on,ie, no changes. The Colour and PG tables on the other hand are both empty at setup,see listing 6-1 for PGT clear,ie VS 4:CLS in BASIC.However,there is a PGT and Colour Table (CT) set up in RAM instead of VRAM,the purpose of this will become apparent shortly.

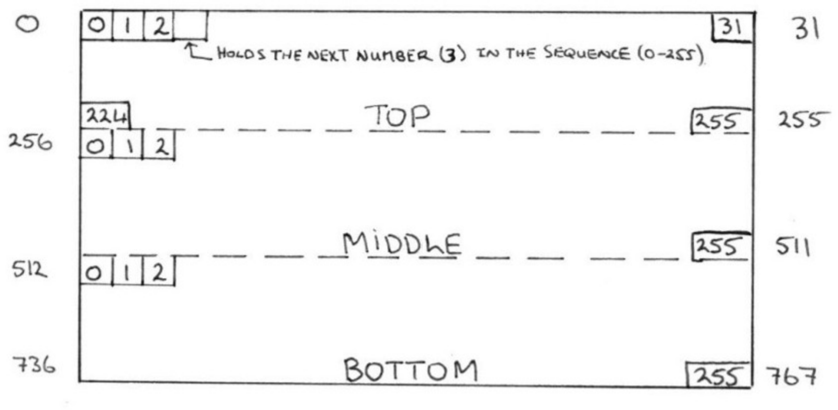


Figure 6-1: The VRAM Pattern Name Table (PNT)

Listing 6-1: This subroutine Mimics the MTX BASIC Command:-
VS 4:CLS. Assumes that the VDP registers have already been set to GII mode.

```
160 CODE

VS4CLS:     PUSH AF                ;
            PUSH HL                ;
            LD HL,(PGTBASE)        ; START OF THE PGT IN VRAM.
            CALL VDPWRTSEL         ; SEE LISTING 3-1.
            LD HL,(PGTLEN)         ; NUMBER OF BYTES TO CLEAR.
CLSPGT:     XOR A                  ; PLACE 0 IN THE 6144 BYTE
            OUT (#01),A            ; TABLE.
            DEC HL                 ; DO UNTIL ALL CLEARED.
            LD A,H                 ;
            OR L                   ;
            JR NZ,CLSPGT           ;
            POP HL                 ;
            POP AF                 ;
            RET                    ; EXIT SUBROUTINE

; SUBROUTINE VARIABLES

PGTBASE:    DW #0000               ;
PGTLEN:     DW #1800               ; 6144 BYTES LONG.

170 RETURN
```

Whenever,we want to print an ASCII character on the VDU,we have to extract the relevant colour and pattern data from RAM. The colour and pattern information are held in the RAM

CT & PGT's respectively. These tables have the same function as the VRAM CT & PGT's. However,as CPU RAM is extremely precious,instead of 3 blocks of 2048 bytes holding the same colour and pattern information,it would make sense only having one table for the 256 patterns and another for the colour information. To extract a ASCII pattern from the RAMPGT and RAMCT ,is a simple matter:

```
----------------------------------------------------
 CPURAMPOSn = RAMASC (or RAMCOL) + ( ASCII Num * 8 )
----------------------------------------------------
```

At this stage we can take this shape (or colour) and store it in a 8-byte temporary buffer and make changes where necessary,ie underlining as in a wordprocessing example. This ability to make pattern/colour changes without affecting the database shape or other on screen characters with the same ASCII number is an extremely powerful asset. When the data is ready,it is sent to the VDP,and directed towards the VRAM PGT and CT at positions which correspond to the CSR X,Y coordinates,see calculation and listing 6-2.

```
---------------------------------------------------------
 CHARPOSn in VRAM PGT = PGTBASE + ( Y * 256 ) + ( X * 8 )
---------------------------------------------------------
 COLPOSn  in VRAM CT  = COLBASE + ( Y * 256 ) + ( X * 8 )
---------------------------------------------------------

 where:
           X           = 0 to 31.
           Y           = 0 to 23.
           PGTBASE     = start of the PGT in VRAM.
           COLBASE     = start of the CT in VRAM.
```

Listing 6-2: Two subroutines which do the above calculations.

```
180 CODE

; this subroutine gets the PGT VRAM address corresponding to
; the MTX BASIC CSR X,Y text coordinates.

GETCHARPOS:PUSH AF               ;
           PUSH HL               ;
           LD HL,(XY)            ; H=Y AND L=X
           LD A,L                ;
           SLA A                 ;
           SLA A                 ;
           SLA A                 ; A=X*8
           LD L,A                ; L RANGE =  #00 TO #FF
           LD A,H                ;
           SLA A                 ; A=Y*256
           LD H,A                ; H RANGE = #00 TO #17.
           LD (CHARPOSn)         ; SAVE IT
           POP HL                ;
           POP AF                ;
           RET                   ; EXIT SUBROUTINE
```

```
; subroutine variables

X:          DS 1                  ; HOLDS THE CSR X COORDINATE
Y:          DS 1                  ; HOLDS THE CSR Y COORDINATE.
CHARPOSn:   DS 2                  ; WHERE CHAR TO BE STORED IN PGT

; this subroutine takes the above CHARPOSn and adds 8192 to
; get the VRAM colour address corresponding to this CSR X,Y.

GETCOLPOS:  PUSH HL               ;
            PUSH DE               ;
            LD HL,(CHARPOSn)      ; GET PGT POSITION.
            LD DE,8192            ; AND ADD TO 8192 TO GET THE
            ADD HL,DE             ; COL POSITION.
            LD (COLPOSn),HL       ; SAVE IT
            POP DE                ;
            POP HL                ;
            RET                   ; EXIT

COLPOSn:    DS 2                  ; WHERE THE COL TO BE STORED.

190 RETURN
```

The CHARPOSn formula requires an explanation as to how it was derived. The CSR X,Y coordinate system has X= 0-31 and Y= 0-23. The bit mapped resolution is 256x192. To calculate the X-dot displacement is X * (256/32=8). Each screen row is 256 dots wide and therefore,to move the Y pointer down , involves moving Y by 256 dots at a time,ie Y*256.

For example,to load a character pattern at CSR 11,10,would relate to a PGT position of 256*10 + (11*8) = 2648. The shape would be loaded at 2648-2655. This VRAM address also tells us which third of the PGT , the character will be loaded at: TOP <2047 ; MIDDLE >2048 & <4095 ; BOTTOM >4096. In this example,the shape is located in the Middle section.

The above mechanism will be repeated,whenever we have to print a character on the VDU,whilst in Bit mapped mode. Therefore,any position in the PGT can be loaded with any 8x8 character shape. This shape will become unique to this screen location (or PGT position). In this way,when we change an ASCII character slightly as required for a WYSIWYG wordprocessor,only the shape at the desired screen coordinate (or PGT position) is changed,all other ASCII characters with the same ASCII number are left untouched. Obviously,writing a WYSIWYG wordprocessor is more complicated than this,but the principle is the same.

I have produced a subroutine flowchart of Listing 5-2 , which highlights the TEXT mode display mechanism and along side of this I have included the Bit mapped TEXT display mechanism, see flowchart at end of this chapter.

6.3 <u>The Bit Mapped Mechanism wrt Plotting and Drawing</u>

In the previous section,we were dealing with 8x8 characters which were displayed on the 32x24 TEXT coordinate system. However,when dealing with individual dots as necessary for drawing lines,circles,etc,involves displaying these on the 256x192 dot or cartesian coordinate system. Both the PGT and CT's are organised similar to the TEXT coordinate system in that it works in bytes rather than bits (dots).

When working with screen dots (or VRAM bits),we have to remember that the Z80/VDP handles all information as bytes and NOT as bits. The consequences of this will reflect in the extra processing required to relate cartesian coordinates to VRAM addresses. The process of plotting a point on the VDU,by writing to the PGT in VRAM involves:

1. Calculate the PGT VRAM address to the nearest BYTE.

2. Determine which BIT in the BYTE,gives the exact PLOT coordinates on the VDU.

3. Convert BIT NUMBER to its BIT VALUE,see figure 6-2.

4. READ this PGT VRAM address and get the current SCREEN byte.

5. This byte holds the screen coordinates adjacent and including the one we require. Care must be taken to SET the bit in question , leaving the other bits as before.

6. Take the new SCREEN byte and send it to the same PGT VRAM address as READ earlier. This will echo the new PLOT X,Y DOT on the VDU.

BIT NUMBER	7	6	5	4	3	2	1	0
BIT VALUE	128	64	32	16	8	8	2	1
BIT PATTERN	0	1	0	1	0	0	0	0
SCRN PATTERN		▨		▨				

Figure 6-2: How the Bit & Screen Patterns correspond.

Before I continue with this discussion,it is appropraite that we diverge and briefly mention Three important Z80 assembly language commands which allow us to manipulate individual or multiple bit(s) of the data/screen byte. These commands are OR,AND and XOR. Table 6-1 summarises the effects of ANDing,ORing and XORing bits of information. As we will see later,simply by changing these operator commands,we can produce three new graphic commands. How these commands work can be found in any good Z80 assembly language book,however,I will give examples of these logical operators in the text where applicable.

Table 6-1: The Truth tables of the Logical Operators: AND,OR and XOR.

BIT States of A & B		**Result of the Logical Operations**		
		OR	**AND**	**XOR**
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

The next step in this complex jigsaw,is the conversion of the PLOT X,Y cartesian coordinates to the PGT VRAM addresses. This calculation is not as straight forward as in the other chapters. As already stated,both the Z80 and VDP handle information in BYTE chuncks. For this reason,we can only determine the VRAM address to the nearest byte. To calculate this:

```
------------------------------------------------------------
 LSB of the VRAM address = (INT(X/8) * 8 ) + (7 - Yremainder)
------------------------------------------------------------
 MSB of the VRAM address = 23 - INT (Y/8)
------------------------------------------------------------
```

where:

```
           X      = 0 to 255.
           Y      = 0 to 191.
           Yremainder = the reminder of the calculation: Y/8.
```

Note that,the above VRAM address should be added to PATBASE to get the actual PGT VRAM address. However,as most people set the PATBASE to 0000,we can therefore ignore it.

Now that the correct VRAM address has been calculated,the next step is to calculate which bit of the byte this address refers too, that gives the exact PLOT coordinates:

```
----------------------------------------------------
 BIT Number = Xremainder (ie the remainder of X/8 )
----------------------------------------------------
```

The INT(X/8) and 23-INT(Y/8) (the reason for the 23- ,is because of the different origins of the TEXT and Cartesian systems ,see figure 6-3) values give the TEXT coordinates, see previous section. The Xremainder and Yremainder values of X/8 and Y/8 respectively,both give a number between 0 & 7.

These numbers refer to the bit column or BIT Number and row byte respectively,see figure 3-1, that makes up this 8x8 block of graphical information for the TEXT coordinate positions. You should keep a note of this fact as it may help you write a WYSIWYG wordprocessor or more importantly a DTP.

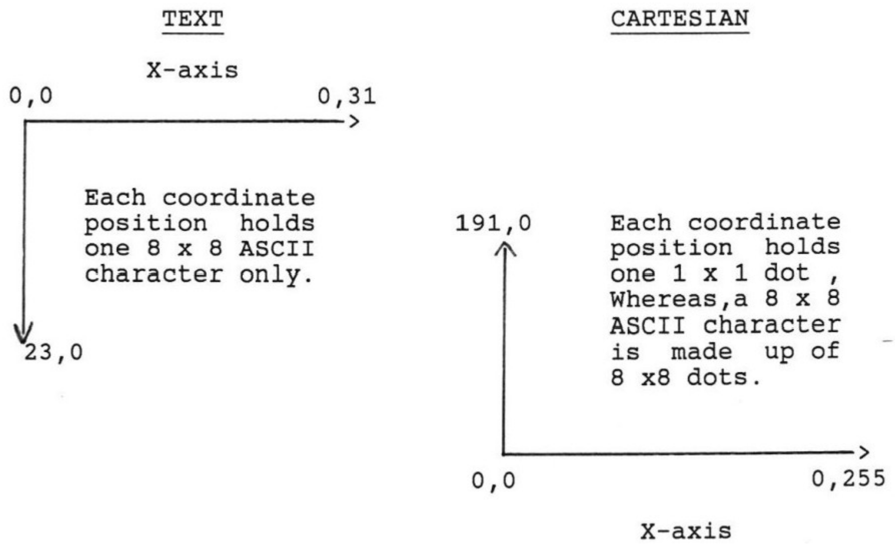

TEXT
X-axis
0,0
0,31
Each coordinate position holds one 8 x 8 ASCII character only.
23,0
CARTESIAN
191,0
Each coordinate position holds one 1 x 1 dot , Whereas,a 8 x 8 ASCII character is made up of 8 x8 dots.
0,0
0,255
X-axis

Figure 6-3: The two coordinate systems used by the VDP: TEXT (32x24) and CARTESIAN (256x192).

We can now pin-point the exact position in the PGT that corresponds to the PLOT X,Y coordinates. However,we will have to just set the BIT Number in question and leave all other bits in the byte untouched. As already stated,this is easily acheived by OR ing the old screen byte with the BIT Value,see Figure 6-2. For example:

```
current Byte pattern   :  0  0  1  0  0  1  1  1  =  39

Bit 6 is to be set     :  0  1  0  0  0  0  0  0  =  64
-------------------------------------------------------
OR result              :  0  1  1  0  0  1  1  1  = 103
-------------------------------------------------------
```

Therefore,the old screen byte of 39 will be overwritten with 103. The next listing includes two subroutines: CARTESIANXY and PLOTXY. The first of which performs the above calculations and the PLOYXY subroutine,mimics the MTX BASIC command PLOT X,Y.

Listing 6-3: The following code mimics the MTX BASIC command PLOT X,Y. Where X = 0 - 255 and Y = 0 - 191.

```
1800 CODE

; When you divide by 8 (= 3 SRL A's) then we are shifting the
; lower 3 bits [ 0-7 ]  out of the byte. These 3 bits make up
; the remainders.The simplest way of getting the remainder is
; to mask  all  bits  except the first three and the new byte
; value will reflect the  remainder. This is  acheived in Z80
; by  ANDing the  X  or Y value with  7 ,the result being the
; Xremainder and Yremainder's respectively.

; MAIN PROGRAM

           LD HL,GIIBITMODE       ; SET VDP TO BIT MODE.
           CALL VDPREGSET8        ;
           LD HL,#1001            ; Y = 16 AND  X= 1
                                  ; THESE VALUES CAN BE CHANGED
           CALL PLOTXY            ; PLOT X,Y
           RET                    ; END PROGRAM


; SUBROUTINES.

CARTESIANXY:PUSH BC               ; HL HOLDS THE X & Y COORDs.
                                  ; ON CALLING THIS SUBROUTINE.
           XOR A                  ;
           LD A,L                 ; L=X-COORDINATE
           SRL A                  ; DIVIDE BY 8
           SRL A                  ;
           SRL A                  ;
           AND A                  ; INT (X/8)
           SLA A                  ; MULTIPLE BY 8
           SLA A                  ;
           SLA A                  ;
           LD C,A                 ; INT (X/8) * 8

           XOR A                  ;
           LD A,H                 ; H=Y-COORDINATE
           SRL A                  ; DIVIDE BY 8
           SRL A                  ;
           SRL A                  ;
           AND A                  ; INT(Y/8)
           LD B,A                 ;
           LD A,23                ;
           SUB B                  ;
           LD (MSBPGTADDR),A      ; SAVE 23-(INT(Y/8)).

           LD A,H                 ; H=Y-COORDINATE
           AND 7                  ; GIVES YREMAINDER.
           LD B,A                 ;
           LD A,7                 ;
           SUB B                  ; 7-YREMAINDER

           ADD A,C                ; INT(X/8)*8 + (7-YREMAINDER)
           LD (LSBPGTADDR),A      ; SAVE IT.
```

```
GETBITNUM: LD A,L                  ; L=X-COORDINATE
           AND 7                   ; XREMAINDER
           LD (BITNUMBER),A        ; SAVE IT.

           LD HL,(LSBPGTADDR)      ; HL=PGT VRAM ADDRESS
           CALL VDPREADSEL         ; READ THE BYTE AT HL.
           IN A,(#01)              ; THIS STORES THE BYTE IN A.
           POP BC                  ;
           RET                     ; A HOLDS THE SCREEN BYTE
                                   ; HL POINTS TO PGT POSITION.


PLOTXY:    PUSH AF                 ;
           PUSH BC                 ; GET VRAM ADDRESS AND THE
           PUSH HL                 ;
           CALL CARTESIANXY        ; BIT NUMBER.
           LD C,A                  ; C=THE OLD SCREEN BYTE.
           CALL VDPWRTSEL          ; TELLS VDP TO WRITE TO THE
                                   ; PGT VRAM ADDRESS.
           LD A,(BITNUMBER)        ; A=BITNUMBER
           LD HL,DOTVALUE          ; NOW CONVERT THE BIT NUMBER
           ADD A,L                 ; INTO ITS CORRESPONDING
           LD L,A                  ;
           LD A,(HL)               ; A=BIT VALUE.

           OR C                    ; CHANGE BIT NUMBER ONLY AND
           OUT (#01),A             ; LEAVE OTHER BITS UNCHANGED
           POP HL                  ;
           POP BC                  ; AND SEND TO THE PGT/SCREEN
           POP AF                  ;
           RET                     ; EXIT

VDPWRTSEL:                         ; SEE LISTING 3-1

VDPREGSET8:                        ; SEE LISTING 4-2

VDPREADSEL:PUSH AF                 ;
           LD A,L                  ; SEND LSB OF ADDRESS TO VDP
           OUT (#02),A             ;
           LD A,H                  ; SEND MSB OF ADDRESS TO VDP
           AND #3F                 ; SELECT READ VRAM.
           OUT (#02),A             ;
           POP AF                  ; VDP READY FOR READING VRAM
           RET                     ;


; PROGRAM VARIABLES


LSBPGTADDR:DS 1                    ;
MSBPGTADDR:DS 1                    ;
BITNUMBER: DS 1                    ; HOLDS THE BIT NUMBER

DOTVALUE:  DB 128,64,32,16,8,4,2,1
GIIBITMODE:DB #02,#C2,#0F,#FF,#03,#7E,#07,#16


1810 RETURN
```

```
Save as:
        SAVE "PLOTXYTXT"                      (tape users)
        DISC (or USER) SAVE "PLOTXY.TXT"      (disc users)
```

Now add:

```
10 GOSUB 1810
20
30 GOTO 30
```

When this program is RUN,a dot at cartesian coordinates 1,16 will be plotted on the Graphics screen. You can try other coordinates by changing the HL value in the program main section. To test,that the dots match up with the MTX BASIC command PLOT X,Y,add at line 20 PLOT 2,16. This will plot a dot at the next X-coordinate. NB: if the dots are not adjacent then check the above code.

As mentioned already,by using the other logical operators: XOR and AND,you can provide two new functions based around the above PLOTXY code. The two new commands are : POINTXY ,see listing 6-4, and TOGGLEXY,see listing 6-5. Those of you who are proficient with BASIC will recognise that BASIC has a POINT X,Y commmand also. Both,return either a 0 or 1 depending on whether the screen position at X,Y is reset or set. The TOGGLEXY subroutine as its name suggests,toggles the dot at X,Y. If position X,Y is in the reset state,then it will be set and vice versa for the set condition.

Listing 6-4: This mimics the BASIC command: POINT X,Y. Useful command for detecting collisions in Arcade games.

```
1900 CODE


;program main

           LD HL,GIIBITMODE            ;
           CALL VDPREGSET8             ; SET TO GII BIT MODE.
           LD HL,#8010                 ; X=16 AND Y=128
           CALL POINTXY                ; SEE IF DOT SET.
           LD A,(POINTSTATUS)          ; 0=OFF 1=ON.
           CP 0                        ; IS IT OFF?
           JR Z,PTOFF                  ; GOTO THE PTOFF SUBR.


PTON:      INSERT CODE HERE FOR PTON.


PTOFF:     INSERT CODE HERE FOR PTOFF.


; SUBROUTINES
```

```
POINTXY:    PUSH AF                     ;
            PUSH BC                     ;
            PUSH HL                     ;
            CALL CARTESIANXY            ; RETURNS A=SCREEN BYTE
            LD C,A                      ; C=SCREEN BYTE.
            LD A,(BITNUMBER)            ; GET THE BIT VALUE.
            LD HL,DOTVALUE              ;
            ADD A,L                     ;
            LD L,A                      ;
            LD A,(HL)                   ;
            AND C                       ; TEST BIT,IF EQUAL
            JR Z,POINTON                ; THEN Z SET.
POINTOFF:   XOR A                       ; SET FLAG TO ZERO.
            JR STOREPOINT               ; SAVE IT
POINTON:    LD A,1                      ; SET FLAG TO ONE.
STOREPOINT: LD (POINTSTATUS),A          ; STORE FLAG VALUE.
            POP HL                      ;
            POP BC                      ;
            POP AF                      ;
            RET                         ; EXIT

CARTESIANXY:                            ; SEE LISTING 6-3

VDPWRTSEL:                              ; SEE LISTING 3-1

VDPREGSET8:                             ; SEE LISTING 4-2

VDPREADSEL:                             ; SEE LISTING 6-3

; PROGRAM VARIABLES

POINTSTATUS:     DS 1                   ;
LSBPGTADDR:      DS 1                   ;
MSBPGTADDR:      DS 1                   ;
BITNUMBER:       DS 1                   ;

DOTVALUE:        DB 128,64,32,16,8,4,2,1
GIIBITMODE:      DB #02,#C2,#0F,#FF,#03,#7E,#07,#16

1910 RETURN
```

Save as:

```
            SAVE "POINTXYTXT"                       (tape users)
            DISC (or USER) SAVE "POINTXY.TXT" disc users)
```

Listing 6-5: This subroutine toggles the dot at X,Y. This command is ideal for DTP and CAD.

```
2000 CODE

;program code

            LD HL,GIIBITMODE            ; SET VDP TO GII MODE
            CALL VDPREGSET8             ;
            LD HL,#7F80                 ; X=128 AND Y=127.
            CALL TOGGLEXY               ; SWITCH THE DOT AT
            RET                         ; X,Y ON OR OFF.
```

```
; SUBROUTINES

TOGGLEXY:   PUSH AF                 ;
            PUSH BC                 ;
            PUSH HL                 ;
            CALL CARTESIANXY        ;
            LD C,A                  ; C=OLD SCREEN BYTE.
            CALL VDPWRTSEL          ;
            LD A,(BITNUMBER)        ; GET BIT VALUE.
            LD HL,DOTVALUE          ;
            ADD A,L                 ;
            LD L,A                  ;
            LD A,(HL)               ;
            XOR C                   ; TOGGLE BIT
            OUT (#01),A             ; SEND NEW BYTE TO SCRN
            POP HL                  ;
            POP BC                  ;
            POP AF                  ;
            RET                     ;

CARTESIANXY:                        ; SEE LISTING 6-3

VDPWRTSEL:                          ; SEE LISTING 3-1

VDPREGSET8:                         ; SEE LISTING 4-2

VDPREADSEL:                         ; SEE LISTING 6-3

; PROGRAM VARIABLES

LSBPGTADDR:DS 1                     ;
MSBPGTADDR:DS 1                     ;
BITNUMBER:  DS 1                    ;
DOTVALUE:   DB 128,64,32,16,8,4,2,1
GIIBITMODE:DB #02,#C2,#0F,#FF,#03,#7E,#07,#16

2010 RETURN
```

Save as:

```
            SAVE "TOGGLEXYTXT"                          (tape users)
            DISC (or USER) SAVE "TOGGLEXY.TXT"          (disc users)
```

Now add this:

```
10 GOSUB 2000
20 GOTO 20
RUN <RET>
```

This will plot a point at 128,127. However,if you press <BRK> and re-RUN the code,the previously on dot will be knocked off.

Finally,drawing lines and circles is a simple matter and I refer you to an excellent article by M.Parlour,see PCW ,September 1987,pages 130-135. Drawing lines requires a mathematical algorithm to calculate the X,Y coordinates along the length of the line. These coordinates are then Plotted using a subroutine like PLOTXY.

6.4 Colour and the Bit Mapped Mode

In GII mode,the colour table has been expanded from 1 byte in TEXT mode,to 32 bytes in GI mode to 6144 bytes in GII mode. This enhanced colour resolution can be addressed as in GII TEXT,GII BIT TEXT and GII BIT Graphics modes. The only difference occuring in the latter mode where only row bytes of graphics can be coloured as opposed to column bits. The reason why is explained next.

The VDP has a 16 colour palette ,numbered 0 to 15. Each dot on the screen can either be on or off. All ON dots will be coloured according to the INK colour and all OFF dots will be in the PAPER colour. To select a colour,whether INK or PAPER ,from a palette of 16 ,can only be defined by 4 bits,as 4 bits gives 16 different patterns,with each pattern corresponding to one of the 16 colours. Therefore,as shown in chapter 3,section 3.3,one byte holds both the INK (msn) & PAPER (lsn) information.

This colour information ,sets the colour for all 8-bits (dots) in a byte. If it was possible to allow full individual colour addressing,ie a INK/PAPER colour for each dot on the screen,then we would require a CT of 48k. Because 1 byte is required for the INK/PAPER colour,a screen resolution of 256x192 (or 48k of dots) will require 1 colour byte per dot. Because of this colour limitation,multicoloured pie charts cannot be drawn properly because of colour clashing. However,by carefully placing coloured graphics or text on the screen,can avoid the above problem.

The System used by the BBC micro,would have been more satisfactory. By reducing the colour palette,the BBC system could offer the user increasing dot and or colour resolution and vice versa. For instance,by reducing the VDP palette to 4 colours would increase the colour addressability from 2 colours per byte to 4 different colours per byte.

Listing 6-6 ,mimics the BASIC commands:VS 4:PAPER p:INK i:CLS This utility can be successfully adapted to provide windowing capabilities. Why not,set yourself this task,as it shouldn't be beyond you with your new found knowledge. PS:use the MTX BASIC commands as a guide.

Listing 6-6: This code mimics: VS 4: PAPER p:INK i:CLS

```
2100 CODE

; PROGRAM CODE

           LD HL,GIIBITMODE           ; SET VDP TO BIT MODE
           CALL VDPREGSET8            ;

CLSVS4SCR: LD HL,0000                 ; START OF PGT IN VRAM.
           CALL VDPWRTSEL             ; TELL VDP
           LD BC,6144                 ; LENGTH OF PGT & CT
           PUSH BC                      ; SAVE CT LENGTH
           CALL CLSPGT                  ; CLS PGT OR SCREEN.

INKPAPSET: LD HL,8192                   ; START OF CT IN VRAM.
           CALL VDPWRTSEL               ; TELL VDP
           LD HL,#010E                  ; INK=BLACK,PAPER=LT RED
           CALL GETCOL                  ; GET COLOUR BYTE
           POP BC                       ; RESTORE CT LENGTH
           CALL SETINKPAP               ; SET SCREEN COLOURS.
           RET                          ; EXIT

; SUBROUTINES

VDPWRTSEL:                              ; SEE LISTING 3-1

VDPREGSET8:                             ; SEE LISTING 4-2

GETCOL:                                 ; SEE LISTING 3-2

SETINKPAP: LD A,(COL)                   ; A = COLOUR BYTE
           EX AF,AF'                    ; SAVE IN AF'
           PUSH AF                      ; SAVE OLD AF'
           JR CLSLOOP                   ; NOW SET SCREEN COLOUR

CLSPGT:    XOR A                        ; SEND 0 TO BLANK PGT
           EX AF,AF'                    ; SAVE IN AF'
           PUSH AF                      ; SAVE OLD AF'
           JR CLSLOOP                   ; CLEAR SCREEN

CLSLOOP:   EX AF,AF'                    ; GET BLANK OR COLOUR
           OUT (#01),A                  ; SEND TO PGT OR CT
           DEC BC                       ; DECREASE LENGTH BY 1
           EX AF,AF'                    ; RESAVE BLANK/COLOUR
           LD A,B                       ; CHECK LENGTH NOT ZERO
           OR C                         ;
           JR NZ,CLSLOOP                ; DO UNTIL ALL 6144 BYTES
           POP AF                       ; RESTORE OLD AF'
           EX AF,AF'                    ;
           RET                          ; RETURN


; PROGRAM VARIABLES

COL:       DS 1                         ; HOLDS COLOUR BYTE.

GIIBITMODE:DB #02,#C2,#0F,#FF,#03,#7E,#07,#16


2110 RETURN
```

Save as:

```
           SAVE "VS4CLSIPTXT"                           (tape users)
           DISC (or USER) SAVE "VS4CLSIP.TXT:           (disc users)
```

What you need to do is write TEXT or Graphics to VS 4, via BASIC & then call the above code to CLS it with GOSUB 2100.

Addressing the colour when dealing with the BIT mode graphics ,is as for the graphics, except that we don't need to worry about determining which bit in the byte needs setting as the colour byte is the same for all 8-bits in the PGT byte.

There will be many instances where you will want to highlight a piece of TEXT or Graphics. The most common method of doing this is to invert the INK and PAPER colours, ie White on Black becomes Black on White. This is easily done from assembly language, using the following steps:

1. Calculate the CT address in VRAM to be inverted.
2. Tell the VDP to READ this address.
3. READ the appropriate colour byte(s). Graphics = 1 byte and TEXT = 8 bytes to be READ,store in buffer.
4. INVERT all the colour bytes.
5. Tell the VDP to WRITE to the same VDP address as READ.
6. Send the new colour byte(s).

When inverting TEXT,all 8 bytes of colour information will be READ unless you have set them all to the same colour. The listing below demonstrates how to invert the colour.

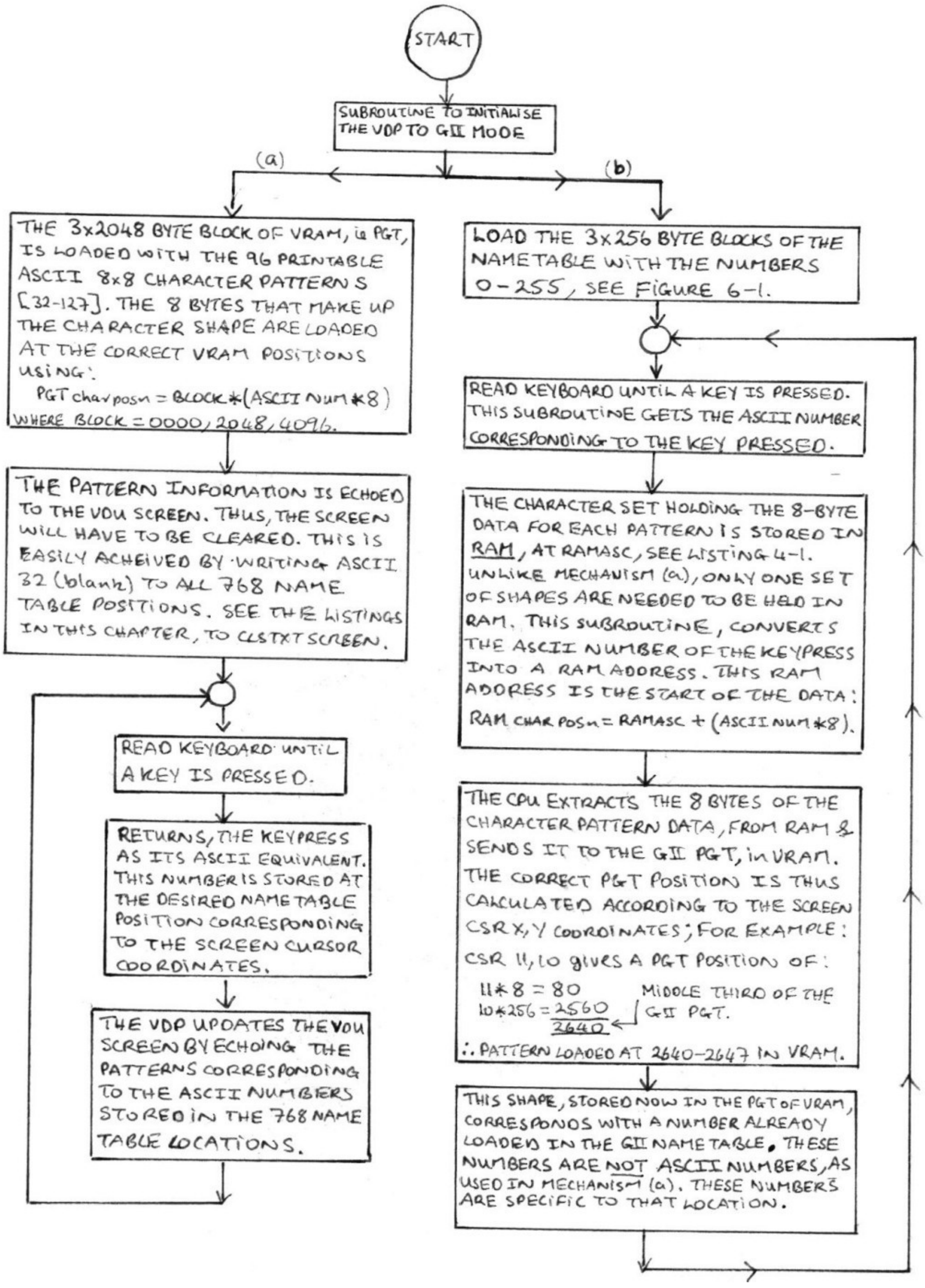
Listing 6-7: Colour Inversion. Assumes VDP set to GII or GI mode.

```
200 CODE

INVCOLBYTE:LD HL,CTVRAMADDR            ; HL=CT VRAM ADDRESS TO
           PUSH HL                     ; READ.
           CALL VDPREADSEL             ;
           IN A,(1)                    ; READ BYTE.
INVERT:    RLCA                        ; SHIFTS THE LSN TO MSN
           RLCA                        ; & THE MSN TO THE LSN.
           RLCA                        ;
           RLCA                        ;
           POP HL                      ; GET VRAM ADDRESS
           CALL VDPWRTSEL              ; AND TELL VDP TO WRITE
           OUT (#01),A                 ; SEND INVERTED COLOUR.
           RET                         ; EXIT

VDPWRTSEL: see LISTING 3-1

VDPREADSEL:see LISTING 6-3

210 RETURN
```

Finally,to set the border colour in the Graphic modes requires writing to VDP 7 register. The Border colour defaults to the colour of the TEXT mode PAPER colour. Therefore,if we set the TEXT mode PAPER colour to the colour we want the BORDER to be and send this to VDP register 7,ie CALL TXTSCRCOL (see listing 3-2) and make sure COL=BORDERCOL.

Figure 6-4: The flowchart below briefly summarises the mechanisms involved in GII TEXT & BIT mapped modes for echoing characters to the VDU.

(a) is the TEXT mechanism.
(b) is the BIT mapped mechanism.



START

SUBROUTINE TO INITIALISE THE VDP TO GII MODE

(a)

THE 3x2048 BYTE BLOCK OF VRAM, ie PGT, IS LOADED WITH THE 96 PRINTABLE ASCII 8x8 CHARACTER PATTERNS [32-127]. THE 8 BYTES THAT MAKE UP THE CHARACTER SHAPE ARE LOADED AT THE CORRECT VRAM POSITIONS USING:
PGT charposn = BLOCK*(ASCII Num*8)
WHERE BLOCK = 0000, 2048, 4096.

THE PATTERN INFORMATION IS ECHOED TO THE VDU SCREEN. THUS, THE SCREEN WILL HAVE TO BE CLEARED. THIS IS EASILY ACHEIVED BY WRITING ASCII 32 (blank) TO ALL 768 NAME TABLE POSITIONS. SEE THE LISTINGS IN THIS CHAPTER, TO CLSTXT SCREEN.

READ KEYBOARD UNTIL A KEY IS PRESSED.

RETURNS, THE KEYPRESS AS ITS ASCII EQUIVALENT. THIS NUMBER IS STORED AT THE DESIRED NAME TABLE POSITION CORRESPONDING TO THE SCREEN CURSOR COORDINATES.

THE VDP UPDATES THE VDU SCREEN BY ECHOING THE PATTERNS CORRESPONDING TO THE ASCII NUMBERS STORED IN THE 768 NAME TABLE LOCATIONS.

(b)

LOAD THE 3x256 BYTE BLOCKS OF THE NAMETABLE WITH THE NUMBERS 0-255, SEE FIGURE 6-1.

READ KEYBOARD UNTIL A KEY IS PRESSED. THIS SUBROUTINE GETS THE ASCII NUMBER CORRESPONDING TO THE KEY PRESSED.

THE CHARACTER SET HOLDING THE 8-BYTE DATA FOR EACH PATTERN IS STORED IN RAM, AT RAMASC, SEE LISTING 4-1. UNLIKE MECHANISM (a), ONLY ONE SET OF SHAPES ARE NEEDED TO BE HELD IN RAM. THIS SUBROUTINE, CONVERTS THE ASCII NUMBER OF THE KEYPRESS INTO A RAM ADDRESS. THIS RAM ADDRESS IS THE START OF THE DATA:
RAM CHAR POSN = RAMASC + (ASCII NUM*8).

THE CPU EXTRACTS THE 8 BYTES OF THE CHARACTER PATTERN DATA, FROM RAM & SENDS IT TO THE GII PGT, IN VRAM. THE CORRECT PGT POSITION IS THUS CALCULATED ACCORDING TO THE SCREEN CSR X,Y COORDINATES; FOR EXAMPLE:
CSR 11,10 GIVES A PGT POSITION OF:

11*8 = 80
10*256 = 2560
2640

MIDDLE THIRD OF THE GII PGT.

∴ PATTERN LOADED AT 2640-2647 IN VRAM.

THIS SHAPE, STORED NOW IN THE PGT OF VRAM, CORRESPONDS WITH A NUMBER ALREADY LOADED IN THE GII NAME TABLE. THESE NUMBERS ARE NOT ASCII NUMBERS, AS USED IN MECHANISM (a). THESE NUMBERS ARE SPECIFIC TO THAT LOCATION.

7.0 Sprites

7.1 Introduction - What is a Sprite?

The TI TMS 9929 handbook describes a sprite as "Special animation orientated patterns that can be made to move rapidly about the screen and change shape with very little programming effort". This is true for all hardware sprites,but not true for software sprites. Hardware sprites as available on the MTX,MSX and Einstein computers,are controlled by a specialised graphics chip - the VDP. Hardware sprites have a number of important features:

(a) Sprites are usually of fixed (same) size,see section 7.5.
(b) Sprite Movement is flicker-free,see section 7.2 .
(c) A sprite can be set to any of the INK colours , but the background has to be transparent,so that the background screen graphics behind the sprite can be seen , see section 7.7.5.
(d) Sprites are on different hardware planes,see section 7.2.
(e) Sprites have a priority associated with them. Coupled with sprites on different planes, it is possible to have 3-D type displays,see sections 7.2 .
(f) Sprites have the ability to appear/disappear and to bleed on and off the VDU picture,see section 7.7.2 and 7.7.3 .
(g) Sprite collision detection . Most specialised graphics processors are able to recognise when two sprites have collided,see sections 7.9.
(h) Sprites are located on the screen according to cartesian coordinates, X and Y , see section 7.7.2 and 7.7.3 .

As you will have gathered from the above list of functions ,the programmer has a powerful Sprite toolbox at his or hers disposal. This toolbox will speed up program design and lead to more stunning and faster arcade games.

I will leave the story of Software Sprites for you to investigate. As a starter - read the article "Graphic Detail" by K.Garroch,PCWeekly,pg 22,issue 16-22 October 1987.

7.2 The Sprite and the Sprite Plane

The MTX,MSX and Einstein computers graphic processor,the VDP,supports 32 hardware sprite planes numbered 0 to 31. Each of these planes can display only one sprite pattern ,selected from a list of 256 from size 0 sprites or from 64 for size 1 sprites,more on this later.

The VDP supports 35 different hardware planes: 32 sprite ,an active pattern plane ,a non-active backdrop plane and an external plane (not considered). A plane is a flat or level surface,analogous to a sheet of paper. Keep this simplistic definition in your head. We will consider what we mean by the pattern and backdrop planes and how these planes are arranged in space.

The whole VDU picture is equivalent to the non-active non-active (non-displayable) backdrop plane. Positioned on top of and in the middle of this backdrop plane is the pattern plane. The pattern plane holds all the TEXT and GRAPHIC patterns held in VRAM,ie its the Graphics screen (256x192 dots) or the TEXT screens (32x24 or 40x24 characters). Both the pattern and backdrop planes are fixed in space. Since the non-active (non-displayable) backdrop plane is bigger than the active (or displayable) pattern plane,you would expect to see the non-active part around the edges of the pattern plane. And this is the case,this region is called the BORDER,which you should all be familar with by now.

Consider the backdrop plane as a sheet of white A3 paper and the pattern plane as sheet of blue A4 paper. Now place the A4 blue plane on top of the white A3 plane,in the middle. The white area is the border region and the blue area is the computer screen. As you can see,the blue area is only covering the backdrop and not overwriting it. This principle is important to comprehend. Therefore,if the blue A4 pattern plane wasn't fixed but was variable,we could move it to any position on the backdrop plane,covering/exposing different areas as we moved it. The backdrop wouldn't be overwritten as this would violate the intergrity of a plane,ie whats on one plane does NOT overwrite but rather HIDES whats on the plane below it.

There are 32 sprite planes of the same dimensions and position in space as the pattern plane. These 32 sprite planes are stacked on top of each other like the slices of bread in a loaf. As we'll see later ,each sprite plane can only display one sprite of a maximium size of 32x32 dots. However the majority of the plane, apart from the area covered by the sprite,is invisible because the background is transparent or see through. Therefore it is probably easier to think of each sprite as a tiny pattern screen which can located anywhere on the main pattern plane,rather than as large invisible sprite planes with one sprite per plane.

As you can see from figure 7-1,that I have numbered the displayable sprites according to there priority or distance from the user. Sprite 0,is closest to the user and threfore,the user will see this sprite first. Therfore sprite 0 has the highest priority with sprite 31 having the lowest sprite priority,ie furthest from the viewer.

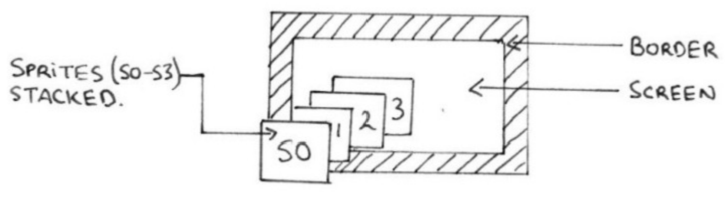


Figure 7-1: Displaced stacked sprites,to give a 3-D type display.

We are able to get this stacked/priority effect because each of these sprites are held on different sprite PLANES. Note that the above figure also tells us that the pattern and backdrop planes are the second lowest and lowest priority planes repectively. The area covered by the sprite is analogous to the area covered by a filing cabinet. When you move the cabinet (ie Sprite) ,you cover another part of the carpet (ie screen) but uncover the previously covered area.

Therefore a sprite plane is a transparent screen which covers the whole VDU picture. There are 32 such planes,stacked on top of each other. Each plane can only display one sprite pattern of maximum size of 32x32 dots,and the sprite can be coloured with any of the 16 colours available. The sprites can move anywhere on the sprite plane freely and flicker free,without affecting other sprites (except when the fifth sprite rule is violated) or affecting the graphics screen on the pattern plane,because the sprites are on different and higher priority planes.

7.3 VRAM Setup

Three VDP registers [R1 , R5 and R6] control the size, position, shape and movement of the sprites. However a Fourth VDP register, R8 ,is also required for checking sprite collisions, fifth sprite rule violations ,see later. This latter register,is READ only,ie can only provide information , whereas, R1,R5 & R6 can be updated or changed. How these VDP registers are used will be covered in the following sections.

7.4 The Sprite Pattern Generator (SPG) Table

The SPG table holds all the shape data for the sprites,just like the TEXT and Graphic VRAM pattern generator tables which store the shape data that make up the ASCII characters or User Defined Graphics,UDG's, respectively. VDP register 6,determines the position of the SPG table in VRAM. The starting address of the SPG table is calculated from:

```
-------------------------------------
 SPGBASE = Register 6 [0-7] * 2048
-------------------------------------
```

VDP register 6 on the MTX is set to 7,which gives the SPGBASE = 7 * 2048 = 14336. This table has a maximium length of 2048 bytes or 2k. For this reason,the SPG table can only start on 2k boundries,see chapter 2. A 2048 byte pattern table corresponds to 256 8x8 sprite shapes or 64 16x16 sprite shapes,see next section.

7.5 The Shape and Size of Sprites

The VDP is limited to only two different sizes of Sprite, either 8x8 or 16x16, see figure 7-2. The size of ALL sprites is selected by altering the status of bit 1 of VDP Register 1,see chapter 2. When bit 1 is 'OFF' (=0) then 8x8 sprite patterns selected, however when this bit is 'ON' (=1) then 16x16 sprites have been selected. Note that,all sprites have to be the same size because only one bit is used to define the sprites size. Individual Sprite size control would require one bit per sprite.

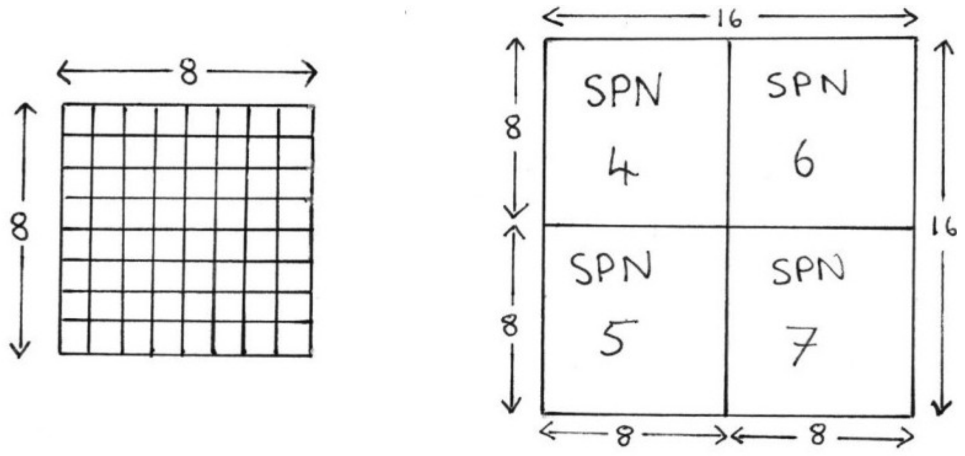


Figure 7-2: Sprite Pattern Grids for size 0 sprites (8x8) and for size 1 sprites (16x16). Size 1 sprites are equivalent to four 8x8 sprite patterns.

As already stated,the Sprite Pattern generator Table is 2048 bytes long. A size 0 sprite is built up from one block of 8 bytes of pattern information. On the other hand,a size 1 sprite is defined by 4 blocks of 8 bytes. As shown in Figure 7-2,the VDP takes 4 smaller size 0 sprites and pieces them together jigsaw style to give 8x8 quadrant. Figure 7-3,shows how the SPG table relates to Sprite Pattern Numbers (SPN). Please note that a sprite pattern number is NOT the same as the Sprite number,more on this latter on.

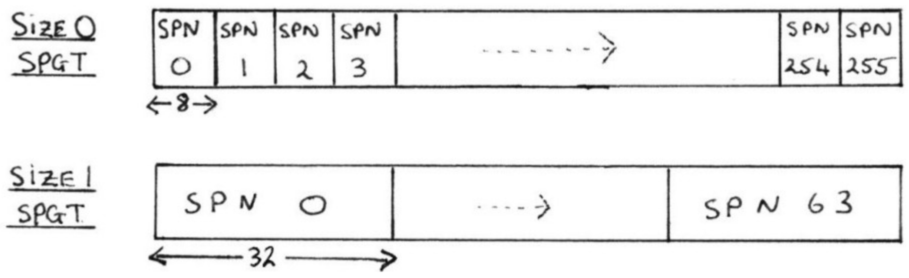


Figure 7-3: The SPG table mapping for 8x8 sprites (size 0) & for 16x16 sprites (size 1).

The number of sprite patterns that are allowed for size 0 in a 2048 byte SPG table is 256 ,ie 2048/8. This number falls to only 64 patterns for size 1 sprites,ie 2048/(4*8). To load the SPG table with the sprite pattern data,is a simple task. First decide on whether its a SIZE 0 or 1 Sprite. Then decide what sprite pattern number you are going to allocate to the data to be loaded in VRAM. Then calculate the correct VRAM address to load the data,using either the SIZE 0 or 1 formula:

```
-------------------------------------------------------------
 SIZE 0 SPGTPOS = SPGBASE + ( Sprite Pattern Number * 8 * 1 )
-------------------------------------------------------------
 SIZE 1 SPGTPOS = SPGBASE + ( Sprite Pattern Number * 8 * 4 )
-------------------------------------------------------------
```

where SPN = 0 - 255 for SIZE 0 and 0 - 63 for SIZE 1 sprites and SPGBASE is the starting address of the Sprite Generator table in VRAM - on the MTX SPGBASE=14336.

7.6 Magnification

Before we leave the topic of sprite sizing,let us consider magnification. Magnification ,enlarges the sprite to double its onscreen size. Therefore,a 8x8 sprite will be magnified to 16x16 and a 16x16 sprite to 32x32. However,although the onscreen sprite appears to have doubled in size,it hasn't doubled in resolution,ie no more data is needed to define a 32x32 sprite than a 16x16. All the VDP has done is take each bit of the sprite data of the 16x16 and copied it once in both the horizontal and vertical directions and then redisplayed it. Therefore the a 1x1 screen dot would become 2 x 2 in size.

The VDP uses only 1 bit of Register 1 (bit 0) to control ALL the sprites magnification. Therefore,you cannot magnify individual Sprites seperately. Once the Magnification bit has been set or reset,then ALL the onscreen sprites will either double or remain unchanged respectively,refer chapter 2.

7.7 The Sprite Attribute Table (SAT)

7.7.1 Overview

The VDP reserves one 128 byte section of VRAM for the Sprite Attribute Table (SAT). The location of this table in VRAM is determined by the contents of VDP register 5,as shown in the following calculation:

```
---------------------------------------
 SATBASE = Register 5 [ 0-127 ] * 128
---------------------------------------
```

This 128 byte table is subdivided into 32 blocks of 4 bytes (32*4=128). Each of these blocks contains the four sprite attribute bytes which are necessary for controlling, positioning, displaying and colouring the 32 allowable onscreen sprites. A more informative description is given in the forthcoming sections,however,the following summary will suffice for this overview.

1. Y-coordinate or Vertical Sprite Position.
2. X-coordinate or Horizontal Sprite Position.
3. The Sprite Pattern Number (SPN). Remember that this not same as the 32 onscreen sprites.
4. The Sprite Colour and Early clock bit.

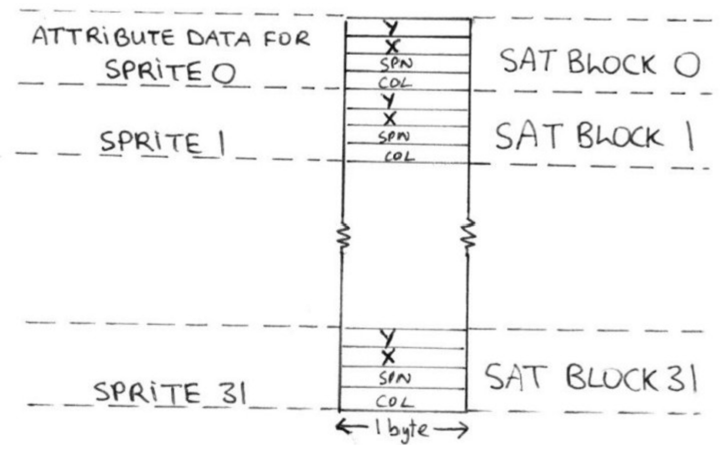
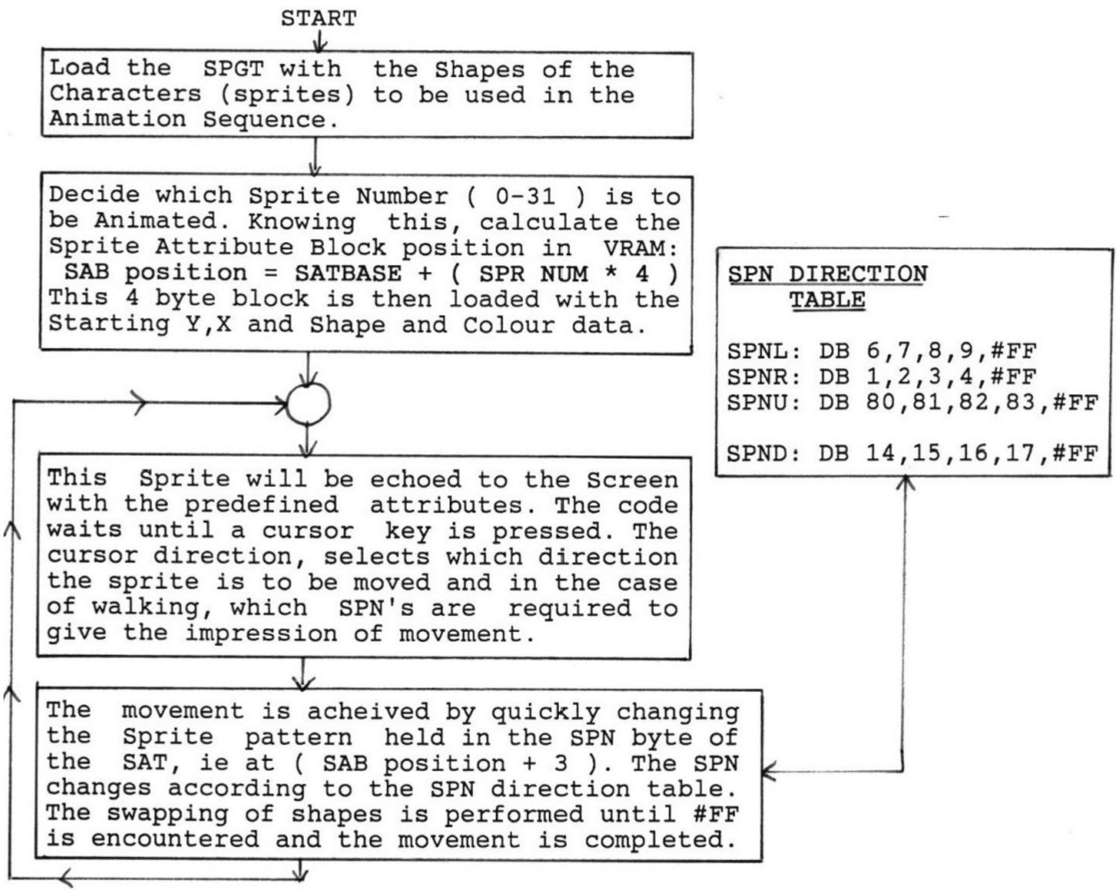


Figure 7-4: The SAT mapping

The above sprite map ,see figure 7-4,allows the programmer increased flexibility. This flexibility is best illustrated by way of a practical example.For example,an arcade explosion sequence as would be used when something was blown up or zapped. Let us consider the process of an object exploding. The object goes from the original unzapped state via a number of breaking up stages,eventually leading to the destroyed (or zapped) object. Therefore, designing an explosion sequence would involve imagining how the object would break up in slow motion ,ie like taking a series of still photographs at points in time as the object was exploding.

We would then overlay these photos onto either a 8x8 or 16x16 grid . Then translate these into data bytes ,ie 8 for 8x8 and 32 for 16x16. Do this for all photos in the sequence. At this stage we should have 4 or 5 different sprite patterns including the starting object and destroyed object. Store these patterns in the SPG table. These patterns are located in the SPG VRAM table at addresses corresponding to the SPN's assigned to them,see section 7.5 for calculations.

Whenever,a sprite blows up, we would, use a simple timing loop,and by switching the SPN of the sprite,acheive a realistic explosion sequence. The timing is crucial to give it that animated look. A similar concept is used in animation,where 4 or 5 different still frames are stored and switched between to give movement. One of the demos at the end demonstrates this technique using a simple countdown. Obviously this wouldn't be a sprite function, however it does graphically demonstrate the above principle. See flowchart below for the programming technique.



START
Load the SPGT with the Shapes of the Characters (sprites) to be used in the Animation Sequence.
Decide which Sprite Number (0-31) is to be Animated. Knowing this, calculate the Sprite Attribute Block position in VRAM: SAB position = SATBASE + (SPR NUM * 4) This 4 byte block is then loaded with the Starting Y,X and Shape and Colour data.
This Sprite will be echoed to the Screen with the predefined attributes. The code waits until a cursor key is pressed. The cursor direction, selects which direction the sprite is to be moved and in the case of walking, which SPN's are required to give the impression of movement.
The movement is acheived by quickly changing the Sprite pattern held in the SPN byte of the SAT, ie at (SAB position + 3). The SPN changes according to the SPN direction table. The swapping of shapes is performed until #FF is encountered and the movement is completed.
SPN DIRECTION TABLE
SPNL: DB 6,7,8,9,#FF
SPNR: DB 1,2,3,4,#FF
SPNU: DB 80,81,82,83,#FF
SPND: DB 14,15,16,17,#FF

Figure 7-5: Overview Flowchart for Animation.

7.7.2 The Sprite Vertical Position

The vertical position byte is the most complex of the attribute bytes to understand and may take one of two readings before the information becomes clear in your mind. Also,it would be advantageous to you if you read up on binary negative number representation before tackling the rest of this section. Also type in Demos 1 and 2,see section 7.10 .

The vertical position or Y-coordinate of a sprite is controlled by the value held in the first byte of the 4-byte attribute blocks. The sprite coordinate system is a hybrid of the cartesian and the TEXT coordinate systems. Therefore,the axis origin is at the top left hand corner of the screen,but the sprite is positioned on the 256x192 dot screen. The Sprite coordinate system is the mirror image of the cartesian coordinate system. The sprite can roam all over the VDU screen.

The cartesian y-axis range is 0 to 191 (bottom to top of screen), but the sprite range is -1 to 190 (top to bottom) The relationship between dot coordinates and sprite coordinates is:

```
------------------------------
SPRYCOORD = 190 - DOTYCOORD
------------------------------
```

A negative number in binary starts from 255 or #FF downwards,it is assumed that when bit 7 is set then this signifies a negative number and the other 7 bits represent the number. For instance, #FF or 255 is = -1 , #FC is -4 ,refer Z80 book. However,as only one negative number is used,the sprite y-coordinate range is 255,0 to 190,ie -1 to 190. Note that a value of Y=191 will place a sprite of the bottom of the screen, RUN Demo1.

Vertical Sprite Bleeding,ie a sprite that can gradually scroll from offscreen onto the VDU picture,is another powerful effect at our fingertips. Sprites can smoothly scroll from an area off picture,onto the VDU picture in dot increments,and if required,then move on down the y-axis,until it scrolls off the bottom of the screen and vice versa for the upward scrolling. This technique is successfully used in games like Agrovator,where the "pacman" type object scrolls off the bottom of the screen and if by magic,scrolls back onto the screen at the top,see Demo 1.

Finally,there are two special conditions that the vertical position byte can tell the VDP to take action over:

1. Y=208 (#D0),when this number is placed in the vertical byte position,then all the sprites below this are made invisible. For example,If we place 208 in the vertical byte of sprite attribute block 5, then sprites 6 to 31 will be made invisible. The other sprites are restored to the screen,when a Y-value between -1 and 190 is put in the Y-byte of attribute block 5. It is important to remember this when sprites scroll up or down to avoid the 208 and 209 conditions.

2. Y=209 (#D1) , as with point 1.,except ALL sprites are made invisible. They are restored when another number other than 208 or 209 is place in the Y-byte that 209 was in. This is what the TI technical manual says,but Demos 1&2 didn't show this !!!

7.7.3 The Sprite Horizontal Position

This is the second byte in each of the 4-byte attribute blocks. This byte holds the X-coordinate information of the sprite. The sprite range is 0 to 255. Again the sprite can move horizontally in dot steps if required. However,the sprites horizontal range or more correctly,horizontal OFFSET isn't as straight forward as one would expect.

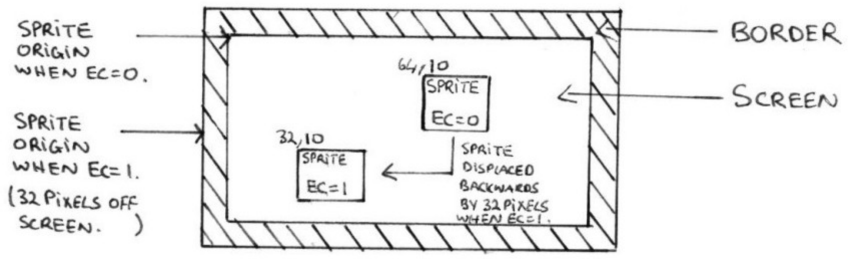

SPRITE ORIGIN WHEN EC=0.
SPRITE ORIGIN WHEN EC=1. (32 PIXELS OFF SCREEN.)
BORDER
SCREEN
64,10
SPRITE EC=0
32,10
SPRITE EC=1
SPRITE DISPLACED BACKWARDS BY 32 PIXELS WHEN EC=1.

Figure 7-6: How the EC bit affects the horizonatal axis.

Figure 7-6,will give you a clue to what I mean. As you will have gathered from this figure,that the origin of the X-axis can be in one or two positions depending on the condition of the early clock bit,which is held in the fourth byte of the block,to be exact ,bit 7. When the EC bit = 0,then the x-axis origin is at the BORDER / SCREEN interface. However,when EC=1 ,the x-axis origin is shifted left by 32 dots.Whenever,the original x-value (ie when EC=0) is < 32 dots,and EC is set to ONE, part or all of the sprite may be shifted off the edge of the screen and that part of the sprite will appear invisible. This latter example is necessary for sprite bleeding from the left hand side of the screen,see Demo 4.

In the EC=1 state,the sprites can bleed on from the left edge of the VDU and when EC=0,the the sprites can bleed off the right edge of the VDU. Therefore,if you want to position the sprite according to the normal dot coordinates of the graphics screen,then make sure EC=0.

7.7.4 The Sprite Pattern Number (SPN)

The sprite pattern number is the specific number of a sprite shape in the SPG table. There are 256 8x8 sprite patterns for size 0 sprites or 64 16x16 sprite patterns for size 1 sprites. When we want to use one of these sprite patterns,we load its number into the SAT,and store it in the 3rd byte of the 4 byte attribute blocks. This tells the VDP when it reads the block information that sprite x (where x=0 to 31),is to have the pattern assigned by the SPN. This principle is used in the TEXT modes.,where an ASCII number (equivalent to the SPN) signifies a specific pattern in the PGT and SPGT's respectively.

7.7.5 The Sprite Colour

A normal colour byte contains the INK and PAPER colours in the MSN and LSN of the colour byte respectively,see chapter 3,figure 3-3. However,as already stated sprites only have INK colours because the PAPER colour has been deliberately set to transparent (hardware set). This gives real time object movement because the background is retained. The LSN (lower 4 bits of the byte) are used to determine the sprites colour from a palette of 16. Bits 6,5,4 are unused and are set to 0. As stated in 7.7.4,Bit 7,is the early clock bit which is used in the positioning of the horizonatal axis origin.

7.8 Sprite VRAM Table Interaction

Figure 7-7 is self explanatory,but to summarise,this figure describes how the SAT and SPG tables interact to produce an onscreen sprite at the desired screen location,in the correct colour and with the right sprite shape.

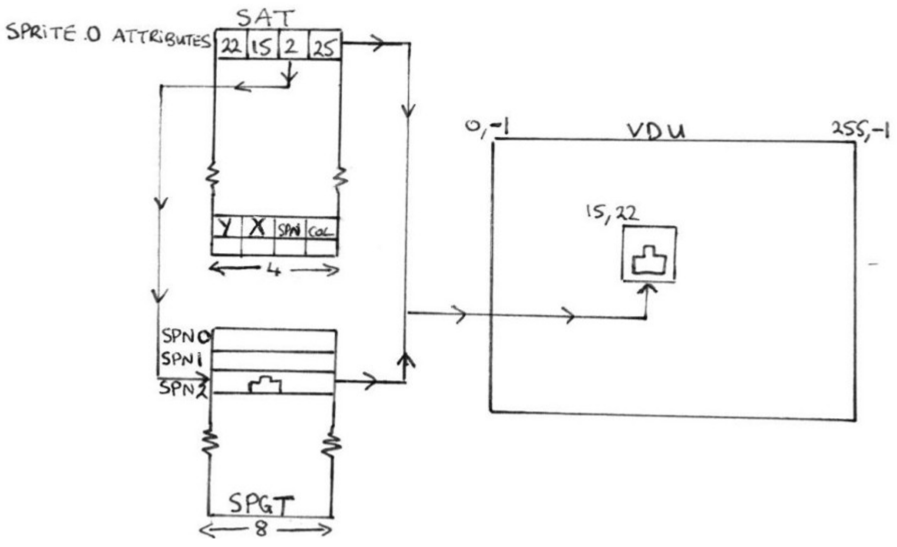


Figure 7-7: Sprite Vram Table Interaction

7.9 The Fifth Sprite Rule and Collision Detection

The VDP imposes a 4 sprite on one Horizontal line only restriction. Even if a fifth sprite temporarily has to cross the same horizontal row,whether partly or fully,the area of the fifth sprite which intersects will disappear,this principle is described graphically in figure 7-8. See also Demos 1 and 2 for this violation.

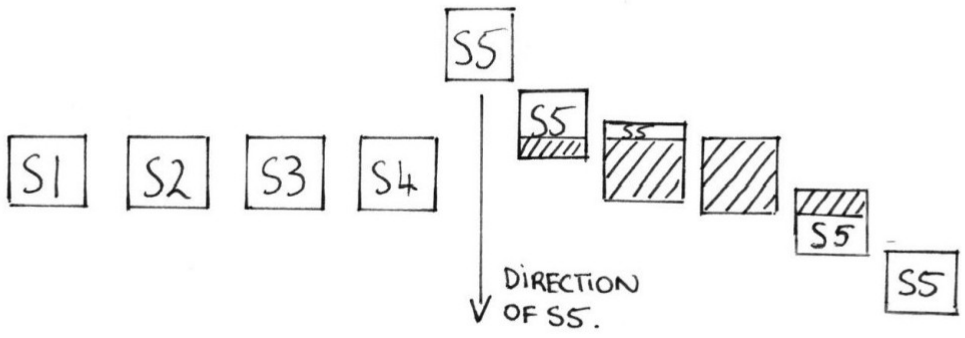


Figure 7-8: Sprite, S5 ,is scrolling downwards,but it has to violate the 5th Sprite Rule. This results in the partial or complete disappearance of S5, as it intersects with the other 4 sprites.

▨ - non visible part of sprite.

▭ - visible of the sprite.

It should be noted that,its the four highest priority sprites which will remain viewable if a 5th sprite rule violation occurs. If you reconsider,the example in figure 7-8,ie swap the roles of S5 and S4,so that S4 is scrolling downwards. The instance that S4 intersects the horizontal line,whether partially or fully,then S5 will completely disappear. This is because,all of S5 is on the same line and therefore all of S5 will be made invisible.

The 5th sprite rule also affects the contents of the read only VDP register 8,when the above rule is violated. Let us consider what the status registers functions are,see Fig 7-9.

a. Reporting the 5th Rule Status.
b. Check for Sprite Collision or coincidence.
c. Check for External Interrupt Signal.

BIT Num:	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	F	5SPR	SC	<---- Sprite Number ----> (0 - 31)				

Figure 7-9: The Read only Status byte (VDP register 8).

where:

F =	interrupt flag.
SC =	sprite coincidence.
5SRF =	5th sprite rule violation (=1). When this happens , the sprite number (0-31) is stored in LSN for the programmer to use.

When the VDP has completed refreshing the VDU picture,it puts F=1 and this allows the VDP to access and read VDP register 8,so that the programmer can check for 5th rule sprite violations,or sprite collisions,etc. When F is reset to 0,the VDP regains control. This is called internal interrupting. There is second interrupt process which occurs when F=1 and when bit 5 of VDP register 1 is also = 1. This is called the external interrupt. This allows the Z80 CPU interrupt mechanism to also interrupt the VDP.

The mechanism by which this rule is violated has already been discussed. The violation is echoed to the VDP by the setting of 5SRF=1 and the sprite number that caused the violation is also stored in the status byte. This is a useful tool for program development,where you want to check to see if sprites crossing each other as in a fast moving arcade game,are suddenly going to disappear because of such a violation.

Finally,sprite coincidence. The SC flag when set to 1,tells us that 2 or more sprites have collided,ie have overlapping dots,this would be set in our 3D display example earlier in this chapter. Note that it is physically impossible for two sprites to collide with each other because they are on different planes,thats why the overlapping dots clause. This collision detection will also check that transparent or invisible sprites,as well as those partially on or off the screen are not overlapping. This type of checking is not really of any use,because it doesn't tell us which sprites or sprite numbers have collided.

7.10 Demonstration Programs for the Sprite Functions

7.10.1 Introduction and Demonstration Listings

In this chapter,I have collected all the demos into one large listing. I have done this because it will minimise typing and it will show up a few interesting points to be wary about when you start using sprites in your own listings. Four simple demos have been devised to highlight the major learning points of this chapter:

1. Fifth Sprite Rule
2. Sprite Priority
3. Sprite Scrolling
4. Sprite Bleeding
5. Sprite Coordinates
6. Size and Magnification
7. Sprite Setup and Programming.

Type in Listing 7-1 and save to disc and tape in the usual way.

Listing 7-1: Sprite Demonstrations: Demos's 1 to 4.

```
10 VS 4:CLS
20 GOSUB 2200
30 STOP

2200 CODE

DEMO:       CALL VDPSETUP           ; SET VDP TO GII MODE
            CALL LOWERPGTSET        ; LOAD THE ASCII SET IN THE BOTTOM
                                    ; THIRD OF THE SCREEN.
            CALL SPGSETUP           ; FILL THE SPG WITH SHAPES.
            CALL WRTXONSCR          ; PUT X= and Y= ON THE SCREEN
DEMOKEY:    XOR A                   ; SCAN THE KEYBOARD FOR KEYPRESSES
            CALL #0079              ;
            JR Z,DEMOKEY            ;
            CP 27                   ; <ESC> TO RETURN TO BASIC
            RET Z                   ;
            CP 128                  ; <F1> FOR DEMO 1
            CALL Z,DEMO1            ;
            CP 129                  ; <F2> FOR DEMO 2
            CALL Z,DEMO2            ;
            CP 130                  ; <F3> FOR DEMO 3
            CALL Z,DEMO3            ;
            CP 131                  ; <F4> FOR DEMO 4
            CALL Z,DEMO4            ;
            JR DEMOKEY              ;

DEMO1:      LD HL,#FF90             ; H=Y-VALUE AND L=X-VALUE OF
            LD (X),HL               ; SPRITE 5 (S5).STORES THESE VALUES
            CALL SATSETUP           ; SET THE SAT VALUES
            LD A,20                 ; DISPLACEMENT IN SAT TO S5 INFO.
            CALL SPRSCRXY           ; DISPLAY S5's COORDINATES ON SCRN
            LD HL,16148             ; SAT POSITION FOR Y IN S5.
            LD (SATPOINTER),HL      ; POINTS TO S5.
DEMO1KEY:   XOR A                   ; SCAN KEYBOARD FOR <ESC> OR <DOWN>
            CALL #0079              ; KEY.
            JR Z,DEMO1KEY           ;
            CP 27                   ; <ESC> TO EXIT TO DEMO LOOP.
            RET Z                   ;
            CP 10                   ; IS THE DOWN ARROW BEEN PRESSED.
            CALL Z,MOVESPRY         ; IF SO MOVE S5 AND UPDATE Y-VALUE
            JR DEMO1KEY             ; END OF DEMO1.

DEMO2:      CALL SATSETUP           ; SET SAT FOR DEMO2.
            LD HL,#FF80             ; THE X & Y-VALUES FOR SPRITE 4,
            LD (X),HL               ; (S4).
            LD A,16                 ; 4*4=16 DISPLACEMENT TO S4 ATTR.
            PUSH AF                 ; POSITION IN VRAM FROM SPGBASE.
            PUSH HL                 ; KEEP TRACK OF THIS INFORMATION.
            CALL GETSATPOS          ; GET THE S4 SAT POSITION.
            CALL VDPWRTSEL          ; TELL VDP WHERE THE START OF S4's
            POP HL                  ; Y-VALUE. RESTORE X & Y
            LD A,H                  ; SEND THE Y-COORDINATE TO THE
            OUT (#01),A             ; SAT TO UPDATE S4'S SCREEN POSITn
            LD A,L                  ; DO LIKEWISE FOR THE X-COORDINATE
            OUT (#01),A             ; ie,PUT S4 AT TOP OF SCREEN
```

```
           POP AF                 ;
           CALL SPRSCRXY          ; DISPLAY S4 COORDINATES ON SCREEN
           LD A,20                ;
           CALL GETSATPOS         ;
           CALL VDPWRTSEL         ;
           LD A,10                ;
           OUT (#01),A            ; PUT S5 ON SAME ROW AS S1,S2 & S3.
           LD HL,16144            ; START OF S4 IN SAT
           LD (SATPOINTER),HL     ; STORE THIS FOR LATER.
           CALL DEMO1KEY          ; MOVE S4 DOWN SCREEN INSTEAD OF
           RET                    ; S5.

DEMO3:     LD HL,(SATBASE)        ; HL=START OF SAT IN VRAM.
           CALL VDPWRTSEL         ; INITIALISE VDP TO POINT HERE.
           LD HL,SATDATA3         ; LOAD THE ATTRIBUTE DATA.
           LD B,4                 ; 4 BYTES TO SEND TO VRAM.
           CALL SENDINFO          ; SEND THEM VIA THIS SUBROUTINE.
D3LOOP1:   CALL COUNTDOWN         ; DISPLAY S5 THEN S4 TO S1 THEN
D3WAIT:    LD B,255               ; BACK TO S5,ie THE COUNTDOWN LOOP
D3WAITLP:  NOP                    ; THIS PIECE OF CODE IS THE DELAY
           PUSH AF                ; LOOP BETWEEN DISPLAYING THE
           POP AF                 ; UPDATED SPN.
           PUSH AF                ;
           POP AF                 ;
           DJNZ D3WAITLP          ;
DEMO3KEY:  XOR A                  ; SCAN KEYBOARD FOR EXIT
           CALL #0079             ;
           CP 27                  ;
           RET Z                  ;
           JR D3LOOP1             ;

DEMO4:     LD HL,(SATBASE)        ; HL=START OF SAT IN VRAM.
           CALL VDPWRTSEL         ; INITIALISE VDP TO POINT HERE.
           LD HL,SATDATA2         ; LOAD THE ATTRIBUTE DATA INTO
           LD B,4                 ; SPRITE 0 POSITION.
           CALL SENDINFO          ; WRITE THE 4 ATTRIBUTES.
           LD HL,#2A00            ; HL=THE X & Y COORDINATES OF S0.
           LD (X),HL              ;
           XOR A                  ; DISPLACEMENT IS ZERO SINCE S0.
           CALL SPRSCRXY          ; DISPLAY THE COORDINATES OF S0.
DEMO4KEY:  XOR A                  ; SCAN THE KEYBOARD
           CALL #0079             ;
           JR Z,DEMO4KEY          ;
           CP 27                  ; <ESC> TO EXIT.
           RET Z                  ;
           CP 25                  ; -> (RIGHT) TO MOVE S0 BY 1 DOT.
           CALL Z,MOVESPRX        ; UPDATE THE S0 AND ITS POSITION.
           CP 132                 ; <F5> TO TOGGLE SPR MAGNIFICATION
           CALL Z,MAGTOG          ;
           CP 133                 ; <F6> TO TOGGLE SPRITE SIZE
           CALL Z,SIZETOG         ;
           CP 134                 ; <F7> TO TOGGLE THE EC BIT.
           CALL Z,ECTOG           ;
           JR DEMO4KEY            ;
```

```
;SUBROUTINES

; A Special note about calling and returning from routines
; which involves scanning the keyboard. Subroutine MOVESPR
; is a good example. This subroutine  is  called  from the
; DEMO4KEY code. When this routine calls MOVESPRX,  reg  A
; =25 (ie move right). However,if MOVESPRX didn't have the
; the lines marked **,then when X increased to 132/133/134
; reg A would equal X on returning from MOVESPRX.This will
; then equal the value for keypresses <F5>/<F6>/<F7> , and
; these subroutines would be called prematurely.So Beware!

MOVESPRX:  PUSH AF              ; **
           LD A,(X)             ; UPDATE THE X-COORDINATE OF S0
           ADD A,1              ; BY 1 DOT AT A TIME.
           LD (X),A             ;
           PUSH AF              ; SAVE THIS VALUE AS ITS NEEDED
           LD HL,16067          ; LATER. UPDATE X ON THE SCREEN
           CALL VDPWRTSEL       ;
           CALL HEXTOASC        ; THIS CONVERTS X TO ASCII FORM
           LD A,1               ; THE X-VALUE FOR S0 IS SATBASE
           CALL GETSATPOS       ; +1.
           CALL VDPWRTSEL       ; SEND NEW X-VALUE FOR S0 TO
           POP AF               ; THIS POSITION IN VRAM.
           OUT (#01),A          ;
           POP AF               ; Reg A= THE LAST KEYPRESS. **
           RET                  ; WHICH WAS 25.

COUNTDOWN: LD HL,(SATBASE)      ; HL=VRAM SAT ADDRESS FOR S0.
           LD A,2               ; BY CHANGING THE SHAPE OF THE
           CALL GETSATPOS       ; OF S0 BY SWAPPING THE SPN AT
           CALL VDPREADSEL      ; SATBASE+2,IS IDEAL FOR THE
           IN A,(#01)           ; ANIMATION TYPE EFFECTS.
           DEC A                ; THIS ROUTINE IS A COUNTDOWN
           CP 0                 ; SIMULATIION. IT IS A REPEAT
           CALL Z,RESSPN5       ; LOOP.
           CALL VDPWRTSEL       ;
           OUT (#01),A          ;
           RET                  ;
RESSPN5:   LD A,5               ;
      RET                       ;

MAGTOG:    LD A,(REGNUM1)       ; REGNUM1 HOLDS THE DEFAULT VALUE
           XOR 1                ; OF ALL SPRITES SIZE AND MAG. BY
           LD (REGNUM1),A       ; TOGGLING BIT 0 OFF AND ON WILL
REGSET:    OUT (#02),A          ; TOGGLE ALL SPRITES MAGNIFICATION
           LD A,#01             ; SEND THE NEW REGNUM1 DATA AND THE
           OR #40               ; CORRECT VDP REG NUMBER (1) TO THE
           OR #80               ; THE VDP.
           OUT (#02),A          ;
           RET                  ;

SIZETOG:   LD A,(REGNUM1)       ; AS FOR MAGTOG EXCEPT TOGGLING THE
           XOR 2                ; SIZE OF ALL SPRITES. ie VDP REG 1
           LD (REGNUM1),A       ; ,BIT 1.
           JR REGSET            ;
```

```
ECTOG:      LD A,(ECFLAG)         ; EC=0 (DEFAULT) OR EC=1. THE EC
            XOR 128               ; BIT IS BIT 7 OF THE 4TH SA BYTE.
            LD (ECFLAG),A         ; THIS IS USED FOR LEFT HAND SPR
            PUSH AF               ; BLEEDING.
            LD A,3                ;
            CALL GETSATPOS        ;
            CALL VDPWRTSEL        ;
            POP AF                ;
            OUT (#01),A           ;
            RET                   ;

VDPSETUP:   LD HL,REGG2MTX        ; INITIALISES THE VDP TO GII MODE.
            CALL VDPREGSET8       ;
            RET                   ;

LOWERPGTSET:LD HL,(ASCBOT3RD)     ; LOADS THE LOWER THIRD OF THE
            CALL VDPWRTSEL        ; PGT WITH THE 96 PRINTABLE ASCII
            LD C,96               ; CHARACTER SHAPES.
            LD HL,RAMASC          ;
BOT3RD:     LD B,8                ;
            CALL SENDINFO         ;
            DEC C                 ;
            JR NZ,BOT3RD          ;
            CALL TXTSCRCLS        ; CLEAR GII SCREEN.
            RET                   ;

SPGSETUP:   LD HL,(SPGBASE)       ; LOADS THE SHAPE DATA OF ALL THE
            CALL VDPWRTSEL        ; SPRITES IN THESE DEMOS. THIS
            LD HL,SPGDATA         ; SUBROUTINE COULD BE ADAPTED TO
            LD B,48               ; LOAD MORE THAN 6 SPRITE SHAPES
            CALL SENDINFO         ; (48/8=6),BY CHANGING THE VALUE
            RET                   ; OF THE B REGISTER.

SATSETUP:   LD HL,(SATBASE)       ; AS FOR SPGSETUP,EXCEPT THAT THE
            CALL VDPWRTSEL        ; SPRITES ATTRIBUTE INFORMATION
            LD HL,SATDATA         ; IS BEING LOADED INTO VRAM.
            LD B,24               ; (24/4=6 SPRITES)
            CALL SENDINFO         ;
            RET                   ;

WRTXONSCR:  LD HL,16065           ; WRITE "X=" AT COORDINATES 1,22.
            CALL VDPWRTSEL        ;
            LD A,88               ; ASCII OF X
            OUT (#01),A           ;
            LD A,61               ; ASCII OF =
            OUT (#01),A           ;

WRTYONSCR:  LD HL,16097           ; WRITE "Y=" AT 1,23.
            CALL VDPWRTSEL        ;
            LD A,89               ; ASCII OF Y
            OUT (#01),A           ;
            LD A,61               ;
            OUT (#01),A           ;
            RET                   ;
```

```
SENDINFO:  LD A,(HL)              ; COPIES THE DATA POINTED TO BY HL
           OUT (#01),A            ; TO VRAM,AT AN ADDRESS SET EARLIER
           INC HL                 ;
           DJNZ SENDINFO          ;
           RET                    ;

GETWRTSCR: ADD A,L                ; HL POINTS TO THE START OF DATA
           LD L,A                 ; & REG A HOLDS THE DISPLACEMENT
           LD A,(HL)              ; OF THE DATA IN THE LIST,WE WANT
           OUT (#01),A            ; TO EXTRACT AND SEND TO VRAM.
           RET                    ;


HEXTOASC:  PUSH HL                ; A HOLDS THE X-COORDINATE TO BE
           PUSH AF                ; CONVERTED TO ITS HEX ASCII
           AND 240                ; EQUIVALENT. NB: NUMBERS IN Z80
           SRL A                  ; ARE STORED IN BINARY AS IT USES
           SRL A                  ; LESS RAM THAN ASCII. THIS CODE
           SRL A                  ; WILL EXTRACT THE 2 PARTS OF A
           SRL A                  ; 8-BIT HEX NUMBER AND FROM THIS
           LD HL,HEXCHARS         ; CONVERTED INTO A DISPLACEMENT
           PUSH HL                ; BETWEEN 0 AND 15. BY LOOKING UP
           CALL GETWRTSCR         ; THE ASCII HEX TABLE,WE CAN GET
           POP HL                 ; PRINTABLE FORMAT EASILY.
           POP AF                 ;
           AND 15                 ;
           CALL GETWRTSCR         ;
           POP HL                 ;
           RET                    ;

GETSATPOS: LD HL,(SATBASE)        ; A HOLDS THE DISPLACEMENT OF THE
           ADD A,L                ; SPRITE ATTRIBUTE TO BE CHANGED
           LD L,A                 ; IN THE SAT.
           RET                    ;

SPRSCRXY:  PUSH HL                ; THIS ROUTINE DISPLAYS THE X & Y
           CALL GETSATPOS         ; COORDINATES OF THE SELECTED
           CALL VDPREADSEL        ; SPRITE,IE S5 OR S4 OR S0. THE
           IN A,(#01)             ; X AND Y HEX COORDINATES ARE HELD
           PUSH AF                ; (POSITIONED) AT 3,22 AND 3,23.
           IN A,(#01)             ;
           LD HL,16067            ;
           CALL VDPWRTSEL         ;
           CALL HEXTOASC          ;
           LD HL,16099            ;
           CALL VDPWRTSEL         ;
           POP AF                 ;
           CALL HEXTOASC          ;
           POP HL                 ;
           RET                    ;

MOVESPRY:  PUSH AF                ; AS FOR MOVESPRX,EXCEPT UPDATING Y
           LD A,(Y)               ;
           INC A                  ;
           LD (Y),A               ;
           PUSH AF                ;
           LD HL,16099            ;
           CALL  VDPWRTSEL       ;
           CALL HEXTOASC         ;
           POP AF                ;
           LD HL,(SATPOINTER)    ;
           CALL VDPWRTSEL        ;
           OUT (#01),A           ;
           POP AF                ;
           RET                   ;


TXTSCRCLS: PUSH AF               ; FOR DESCRIPTION SEE LISTING 5-2.
           PUSH BC               ;
           PUSH HL               ;
           LD HL,(SCRNTYPE)      ;
           CALL VDPWRTSEL        ;
           LD BC,(SCRNLEN)       ;
TXTSCRCLS1:LD A,32               ;
           OUT (#01),A           ;
           DEC BC                ;
           LD A,B                ;
           OR C                  ;
           JR NZ,TXTSCRCLS1      ;
           POP HL                ;
           POP BC                ;
           POP AF                ;
           RET                   ;

VDPWRTSEL:                       ; SEE LISTING 3-1

VDPREADSEL:                      ; SEE LISTING 6-3

VDPREGSET8:                      ; SEE LISTING 4-2

; PROGRAM AND SUBROUTINE VARIABLES

SATPOINTER:DS 2

HEXCHARS:  DB 48,49,50,51,52,53,54,55,56,57,65,66,67,68,69,70,72

SCRNTYPE:  DW 15360
SCRNLEN:   DW 768
REGG2MTX:  DB #02,#C0,#0F,#FF,#03,#7E,#07,#16
NAMEBASE:  DS 2
X:         DB 144
Y:         DB #FF

REGNUM1:   DB #C0                ; DEFAULT=#C0 = SIZE 0 AND MAG 0.
ECFLAG:    DB #0F                ; INK = WHITE AND EC = 0.

SATDATA2:  DB 10,0,0,15
SATDATA3:  DB #80,#A0,#05,#0F
SPGDATA:   DB 48,72,140,148,164,72,48,0      ; 0
           DB 48,80,16,16,16,16,124,0        ; 1
           DB 56,68,4,8,16,32,124,0          ; 2
           DB 120,132,4,60,4,132,120,0       ; 3
           DB 128,128,128,144,144,252,16,0   ; 4
           DB 248,128,128,248,4,4,248,0      ; 5
```

```
SATDATA:    DB 0,0,0,0
            DB 10,80,1,15
            DB 10,96,2,15
            DB 10,112,3,15
            DB 10,128,4,15
            DB #FF,144,5,15

GIISCRN:    DW 15360
SATBASE:    DW 16128
SPGBASE:    DW 14336
ASCBOT3RD:  DW 4352

RAMASC:                      ; SEE LISTING 4-1

2210 RETURN
```

Save as:

```
     SAVE "SPRDEMOSTXT"                    (tape users)
     DISC (USER) SAVE "SPRDEMOS.TXT"       (disc users)
```

7.10.2 Demo 1

Reload listing 7-1 and RUN it. Now select Demo 1 by pressing the function key <F1>. The screen will have cleared and 5 sprites in the form of numbers ,1-5, should have appeared at the top of the display. Sprite 5 will be at the border/screen interface. At the bottom of the screen will be displayed the coordinates of Sprite 5. Using the Down Arrow,to move Sprite 5 down the screen.

As part of Sprite 5 intersects with the horizontal line with Sprites 1 to 4,Sprite 5 will gradually start disappearing until it intersects exactly,when it is completely invisble.However, as it moves further down the screen it will start reappearing. If you keep Sprite 5 moving down the screen,it will then disappear off the bottom of the screen. However,if you keep pressing down,the Y-coordinate will keep increasing and then Sprite 5 will eventually reappear at the top of the screen again.

7.10.3 Demo 2

Reload listing 7-1 and RUN it. Selecting with <F2>. As for Demo 1 ,except that Sprite 4 moves down the screen and Sprite 5 is stationary. This Demo is another example of the Fifth Sprite Rule and also highlights the Priority of it. This time as Sprite 4 intersects with the line with Sprites 1,2,3 and 5,it is Sprite 5 that again disappears even though it is Sprite 4 that is violating the integrity of the line.

7.10.4 Demo 3

Reload listing 7-1 and RUN it. Selecting with <F3>. This example shows the countdown example described in the text earlier on animation. Notice the speed of the countdown process. It is so fast even with a delay loop that is is impossible to disguish the numbers. This example is an example of what power with respect to animation that can be acheived by switching the SPN of a Sprite. I will let you include a few other PUSH's and POP's to slow down the countdown further to see the actual process.

7.10.5 Demo 4

Again Reload listing 7-1 and RUN it. Selecting with <F4>. This example demonstrates horizontal scrolling and bleeding as in used in Demo's 1 and 2. However,this demo also includes 3 toggles:

<F6> :	toggles between SIZE 0 and SIZE 1 sprites. It also shows how a SIZE 1 sprite is made up,see Figure 7-2
<F5> :	This will toggle a Sprites magnification.
<F7> :	This will toggle the EC bit.

It is possible to toggle all three at the same time or in combinations thereof. Feel free to experiment. Another related example is to toggle to SIZE/MAG/EC = 0. Then press <ESC> and press <F1>. Now press <ESC> and then <F4>. Move the sprite across the screen. Sprite 0 will glide above sprites 1 to 5 without destroying there shapes as would have happened in the NON-Sprite modes.

8.0 Screen Dump

8.1 Introduction

The Memotech MTX,MSX and Tatung Einstein are all fortunate to have a centronics parallel interface as part of the computer hardware. I will first take a look at the parallel interface and then provide suitable software to use the parallel interface to give both TEXT and GRAPHIC hardcopy screen dumps.

8.2 Transmitting Data via the Parallel Interface

8.2.1 The Hardware

There are two common ways of transmitting information: in Series or in Parallel. Both systems transmit the binary information of the computer,ie the 0's and 1's ,as electrical signals ,where 0 volts = "0" and 5 volts = "1". The Parallel interface,unlike the rival Serial interface,sends data in BYTE chunks (ie 8 bits at a time). Each bit of the byte data is sent along a different wire simultaneously. Thus 8 bits of information is transmitted along 8 parallel data wires to the printer in one clock (or strobe) pulse,see Figure 8-1.

Whereas,the Serial interface,as its name suggests,sends the same 8 bits of information down one wire at pulse intervals. Therefore it will take 8 clock cycles to send the same information sequentially down one wire. A serial system can send data to printers or other terminals at distances greater than 25 metres. However,a parallel system can only send data at distance of 2 - 5 metres. The greater number of wires used in connecting ,a computer to a parallel printer (17 wires) than a serial (5 wires), is not only costly but the signal is prone to deteriate because of the increasing interference of the wires magnetic fields over distance. This means greater error correction is needed to do the same job.

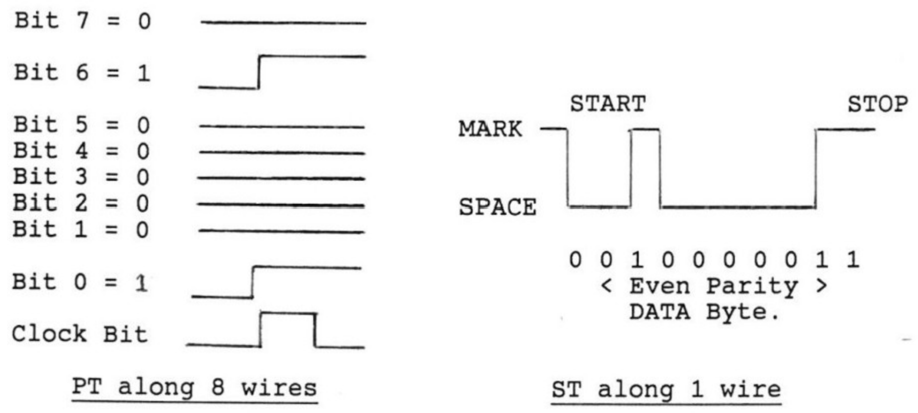


Figure 8-1: Parallel Transmission (PT) & Serial Transmission (ST) of ASCII 65, 01000001 , ie LETTER 'A".

If you turn to page 253 of the MTX owners manual,you will see that the Parallel interface uses more than just 8 of the data wires.

The ACKNLG or acknowledge signal tells the computer that the printer is ready.

The BUSY signal tells the computer that data is currently being transferred along these lines.

The PE or Paper Empty signal is self explanatory.

The ERROR line reports any other faults on the line.

The GND or ground signal indicates what the ground voltage of the computer is.

The SLCT or select signal is used to indicate that the computer is ready to send data to the printer device.

The STROBE or strobe signal (or clock pulse) is very important. It tells the printer that the computer has assembled the 8 bits of information and is ready to send them.

8.2.2 The Software

The Memotech MTX series uses ports 0 and 4 of the Z80 CPU to communicate with the parallel interface device,ie Printer, see pages 248-249 of the MTX Owners Manual. Briefly:-

IN A,(#00) is used to set the STROBE low,ie primed for sending data.

IN A,(#04) is used to read the Printers status. Only the lower 4 bits of the status byte is used:

Bit 0 (d0) = BUSY ; when high (=1) then the line is busy.
Bit 1 (d1) = ERROR ; when low (=0) then error.
Bit 2 (d2) = PE ; when high (=1) then no paper.
Bit 3 (d3) = SLCT ; when high (=1) the printer is selected.

Therefore,for the printer status to be VALID,ie no errors,the status byte will have to have the following bit pattern:

```
d7 d6 d5 d4 d3 d2 d1 d0           (d = data)
0  0  0  0  1  0  1  0 = #0A
```

As you will see,listing 8-1,uses this fact to check the printer status,prior to printing.

OUT (#04),A sends the data held in register A to the printer only when the the strobe is forced low ,using IN A,(#00) and the status byte is Valid (= #0A). Note that after strobing low,you must then force the strobe high using IN A,(#04) after 1 microsecond has elapsed,see listing 8-1. As you will appreciate timing is very important in communication.

Now we have dealt with the commands that are needed to communicate with the parallel interface printer,I will describe with the use of a flowchart,see figure 8-2,how easy it is to transmit data to the printer. Listing 8-1,is the fruits of the flowchart information in figure 8-2. The two subroutines HARDCOPY and CHBRK are the fundamental building blocks of any advanced hardcopy facilities that you may develop latter. These two subroutines are similar to the MTX ROM code located at #0CE3. However,this code is portable among both MTX CPM and normal mode. The principles used can be easily adapted for both Einstein and MSX users.

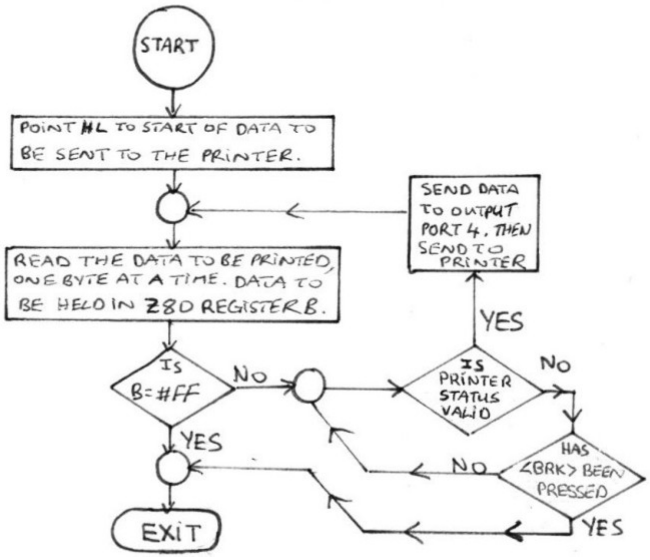

START
POINT HL TO START OF DATA TO BE SENT TO THE PRINTER.
READ THE DATA TO BE PRINTED, ONE BYTE AT A TIME. DATA TO BE HELD IN Z80 REGISTER B.
IS B=#FF
NO
YES
SEND DATA TO OUTPUT PORT 4, THEN SEND TO PRINTER
YES
IS PRINTER STATUS VALID
NO
HAS <BRK> BEEN PRESSED
NO
YES
EXIT

Figure 8-2: Flowchart of Listing 8-1.

Listing 8-1:This routine passes the data held in Register B of the Z80 CPU to the HARDCOPY subroutine. This data will only be sent to the printer if the following two criteria are met: (a) the status byte is valid and (b) the <BRK> key hasn't been pressed,as this terminates the printout.

```
10 GOSUB 2300
20 STOP

2300 CODE

           LD HL,PRINTDATA       ; HL POINTS TO THE DATA TO BE
                                 ; PRINTED. DATA IN ASCII NUM
PRNTLOOP:  LD B,(HL)             ; FORMAT. B SENDS THESE ASCII
           LD A,#FF              ; NUM's TO THE PRINTER UNTIL
           CP B                  ; THE END MARKER (#FF) IS
           RET Z                 ; ENCOUNTERED,THEN EXIT.
           CALL HARDCOPY           ; IS THE PRINTER STATUS VALID
           JR PRNTLOOP             ; CAN ONLY LEAVE THIS EXAMPLE
                                   ; WHEN STATUS IS NOT VALID OR
                                   ; #FF IS ENCOUNTERED OR WHEN
                                   ; THE <BRK> HAS BEEN PRESSED.

HARDCOPY:  NOP                     ; THIS IS THE PRINTOUT CODE.
CHPRTSTAT: IN A,(#04)              ; READ PRINTER STATUS. ONLY
           AND #0F                 ; THE LSN IS NEEDED. IS THIS
           CP #0A                  ; BYTE VALID. IF NOT CHBRK.
           JR NZ,CHBRK             ;
PRINTBYTE: LD A,B                  ; LET A=THE DATA BYTE & HAVE
           OUT (#04),A             ; IT READY TO SEND WHEN THE
           IN A,(#00)              ; STROBE IS LOW. THEN RESET
           IN A,(#04)              ; THE STROBE. THE ASCII CHAR
           RET                     ; SHOULD HAVE BEEN PRINTED.

CHBRK:     LD A,#FE                ; THIS IS THE ROW THE BRK
           OUT (#05),A             ; IS ON,SEE CH9.SELECT IT
           IN A,(#06)              ; READ THIS ROWS STATUS.
           BIT 0,A                 ; IS BRK PRESSED. IF NOT
           JR NZ,CHPRTSTAT         ; CHECK PRINTER STAT AGAIN
           POP HL                  ; THIS REMOVES THE RETURN
                                   ; ADDRESS OF CALL HARDCOPY
                                   ; FROM THE STACK,EXPOSING
           RET                     ; EXPOSING THE RET ADDRESS
                                   ; TO BASIC.

PRINTDATA: DB 84,69,83,84,73,78,71,#FF ; "TESTING"

2310 RETURN
```

Save as:

```
      SAVE "PRINTERTXT                       (tape users)
      DISC (USER) SAVE "PRINTER.TXT"         (disc users)
```

When you RUN the above code the word "TESTING" will be sent to the printer. Pressing <BRK> will abort at any stage,But this listing is too short for <BRK> to respond fast enough because of the printer buffer.

8.3 Screen Types and their Screen Dump Software

8.3.1 Introduction

On the MTX there are 4 main screen formats: GRAPHICS, TEXT , PANEL and NODDY. Fortunately for the MTX programmer,the latter three screen modes are basically TEXT screens. All that the Memotech programmers have done is reconfigure the TEXT screen to act as a Z80 disassembly monitor screen or as a programming NODDY text editor. Therefore,the software needed to produce screen dumps of these screens will be very similar,see next subsection. The GRAPHICS dump is far more complex and involves rotation of screen data,similar to the ASCII ROM character set rotation in chapter 3.0 .

8.3.2 TEXT Only Screen Dump Software

For people unfamilar with the MTX computer,the PANEL screen is a Z80 machine code monitor that complements the built-in Z80 line assembler. This Z80 debugging tool,displays the Z80 register status,a HEX/ASCII memory dump and disassembly. The Z80 monitor program allows the programmer to run selected pieces of code,or even to step through the code,checking how the code affects Z80 registers,flags,memory,etc. NODDY is a text orientated language which is ideally suited to handling text as necessary for databasing,information retrieval,card indexing,diary,etc. The NODDY screens are 39 x 23 (the bottom line is used for NODDY coding). All data and programs are handled by the TEXT editor. The TEXT screen is the default screen on the MTX. It is used for other text applications, BASIC Program Editor ,Error messages ,etc.

If you've read chapter 3,you will realise that the VDP handles all character information according to the ASCII character set protocol. The MTXOS configures the 960 byte TEXT Name Table as a 40x24 TEXT screen. The Name Table stores the ASCII character numbers,ie ASCII 65 is the letter A. The VDP can quickly read this table converting these numbers to the appropriate character shapes held in the TEXT PGT. All this information is echoed to the VDU. The TEXT Name table on the MTX is found at 7168 and the TEXT PGT starts at 6144 in VRAM. The printer uses a similar mechanism when it prints TEXT on the paper as the VDP does for displaying TEXT on the VDU screen.

In normal print mode,the printer receives bytes of information from the computer. This information is a series of ASCII numbers,relating to the TEXT on the screen.The Printer then extracts the correct character shape from its built-in printer character font and prints it onto the paper. Similar to the way the Name Table and PGT interact to give an onscreen presentation. Again ,the reason for only transmitting the ASCII codes; is due to increased speed. As you will see later ,when you have to send actual shape data to the printer,the printing speed greatly decreases.

Thus producing a Hardcopy of the TEXT screen is relatively easy once you know what information is required:

1. Start of the Text Name Table - NAMEBASE (= 7168 on MTX)
2. Length of the Name Table - NAMETABLEN (= 40 x 24 = 960)
3. Listing 8-1 to send the data to the Printer.

Listing 8-2 demonstrates how this can be done in practice.

Listing 8-2: Text Screen Dump

```
10 PRINT " This Text Will  be displayed on the Screen and "
20 PRINT " the code below will Read the TEXT screen and "
30 PRINT " transmit these ASCII codes to the printer to "
40 PRINT " translate into their respective character shapes"
50 PRINT " . The shapes are extracted from the printers own"
```

```
60 PRINT " resident character font."

100 GOSUB 2400
110 STOP

2400 CODE

PRNTTEXT:  LD HL,7168              ; START OF NAME TABLE IN
           CALL VDPREADSEL         ; VRAM. TELL THE VDP TO
           LD DE,960               ; READ THIS 960 BYTE AREA
TEXTPOUT:  IN A,(#01)              ; READ THE SCREEN ASCII
           LD B,A                  ; AND PASS IT TO THE
           CALL HARDCOPY           ; HARDCOPY ROUTINE. THIS
           DEC DE                  ; IS REPEATED UNTIL ALL
           LD A,D                  ; 960 BYTES OR SCREEN
           OR E                    ; LOCATIONS HAVE BEEN
           JR NZ,TEXTPOUT          ; TRANSMITTED TO PAPER.
           RET                     ; EXIT


; SUBROUTINES

HARDCOPY:                          ; SEE LISTING 8-1

CHBRK:                             ; SEE LISTING 8-1

VDPREADSEL:                        ; SEE LISTING 6-3

2410 RETURN
```

Save as:

```
           SAVE "TXTPRINTTXT"                      (tape users)
           DISC (USER) SAVE " TXTPRINT.TXT"        (disc users)
```

When you run the above code,the text on the TEXT screen will be echoed to the printer,only when the printer is on and ready. However,unless you have told your printer to change its column width to that of the TEXT screen (ie 40 columns),you will probabely see two lines of screen text printed on the same line of the printer paper. This nicely leads on to the next section.

8.3.3 Printer Control Codes

The software so far allows us to copy the screen to the paper. However,as the last program has highlighted,we need some way of formatting the TEXT on the paper,ie setting the printer width to the screen width of 40. The Epson and Epson compatible printers provide the user with a vast array of special printing functions,see Table 8-1. These special codes give us total control of the final printed format.

The VDP TEXT orientated screens cannot even match this performance because of the way they were designed. However,programs like Micropro's WORDSTAR,uses keystrokes like CTRL PS to switch underline on. This 3 key press is stored in RAM as a special ASCII code. This code is one of the non-printable ASCII 0-31 or a combination of codes ,see Table 8-1. Whenever,one of these codes is sent to the printer, the printer switches on the desired function like Underline or column width or Bold etc. Listing 8-3 incorporates some of these facilities.

Table 8-1: Some Epson printer control codes.

FUNCTION	**Code for ON**	**Code for OFF**
Enlarge	DB 27,"W",1	DB 27,"W",0
Reduce	DB 15,0,0	DB 18,0,0
Emphasise (bold)	DB 27,"E",0	DB 27,"F",0
Eite/Pica	DB 27,"P",0	DB 27,"P",1
Italic	DB 27,"4",0	DB 27,"5",0
Double Strike	DB 27,"G",0	DB 27,H",0
Underline	DB 27,"-",1	DB 27,"-",0
Unidirection	DB 27,"U",1	DB 27,"U",0
Subscript	DB 27,"S",1	DB 27,"T",0
Superscript	DB 27,"S",0	DB 27,"T",0
Backspace	DB 8,0,0	
LineFeed	DB 27,"A",n	
	*where n=n*1/72"*	
Reset Printer	DB 27,64,0	

Listing 8-3: Text Dump Software with Printer Control Codes.

```
10 PRINT " This Text Will  be displayed on the Screen and "
20 PRINT " the code below will Read the TEXT screen and "
30 PRINT " transmit these ASCII codes to the printer to "
40 PRINT " translate into their respective character shapes"
50 PRINT " . The shapes are extracted from the printers own"
60 PRINT " resident character font."


100 GOSUB 2500
110 STOP


2500 CODE

PRNTTEXT2: LD HL,7168            ; START OF NAME TABLE IN
           CALL VDPREADSEL       ; VRAM. TELL THE VDP TO
           LD DE,960             ; READ THIS 960 BYTE AREA
           LD HL,LINEFEED        ; SET PRINTER TO SCREEN
           LD C,6                ; FORMAT.
           CALL SCTRLCODE        ; SEND TO PRINTER.

TEXTPOUT:                        ; SEE LISTING 8-2
```

```
;SUBROUTINES

HARDCOPY:                        ; SEE LISTING 8-1
CHBRK:                           ; SEE LISTING 8-1

VDPREADSEL:                      ; SEE LISTING 6-3

SCTRLCODE: LD B,(HL)             ; READ THE FUNCTION CODE
           CALL HARDCOPY         ; AND SEND TO PRINTER.
           INC HL                ;
           DEC C                 ;
           JR NZ,SCTRLCODE       ;
           RET                   ;


; PRINTER CONTROL CODE DATA

LINEFEED:  DB 27,"A",12          ; SPACING BETWEEN LINES
COLWIDTH:  DB 27,"Q",40          ; COL WIDTH SET TO 40.

2510 RETURN
```

Save as:

```
           SAVE "TXTPRNT2TXT"                    (tape users)
           DISC (USER) SAVE " TXTPRNT2.TXT"      (disc users)
```

When the above program is RUN,the printed output will be exactly like that of the screen. It is easy to convert the above to give PANEL and NODDY screens - see "MTX PANEL COPY" ,PCWeekly , Vol 6, No 25 and 26 ; see "MTX PRINT SCREEN", Memotechniques ,Vol 4,Issue 5 ; see "MTX SCREEN DUMP" ,Memopad ,vol 010 ,issue 8.

8.4 Graphical Bit Mapped Screen Dumps

The advantages of GII and GI modes are that complex graphic screens can be generated on the screen. However,obtaining hardcopy of such screens involves a lot of processing. And unlike the normal ASCII print mode ,we cannot take advantage of many of the special printer control codes.

8.4.1 Transposition

The MTX,MSX and Einstein Computers treat each byte of screen data as a horizontal sequence of dots,eg d7 d6 d5 d4 d3 d2 d1 d0 = 1 horizontal byte. However,in todays computer market,the Epson printer seems to be the industry standard or at least the most popular one. This causes the programmer a slight technical problem,because the Epson printers bit image mode handles graphical data as a sequence of vertical bytes rather than the screen horizontal byte format,see Figure 8-3.

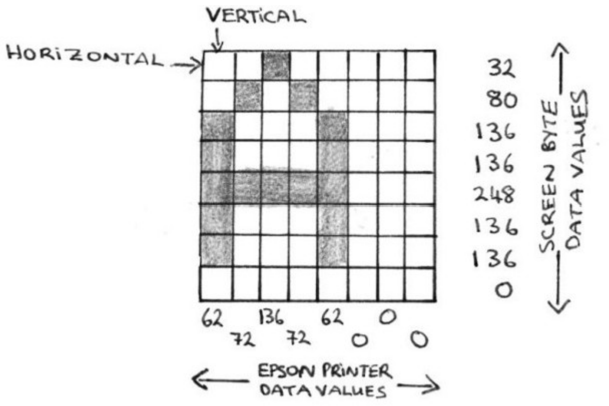


Figure 8-3: Vertical and Horizontal data formats

This means that we have to take the 8 horizontal bytes that makes up a 8x8 graphical pattern and rotate each of them,a bit at a time,to obtain the correct vertical format before sending to HARDCOPY subroutine. The technique of twisting 8 screen bytes of an alphanumeric character is called Transposition. If you think back to chapter 3.0,in particular listing 3-1,you will see that we have performed the opposite of transposition. Figure 3-1 shows the reverse of transposition and figure 8-4 shows the process of transposition.

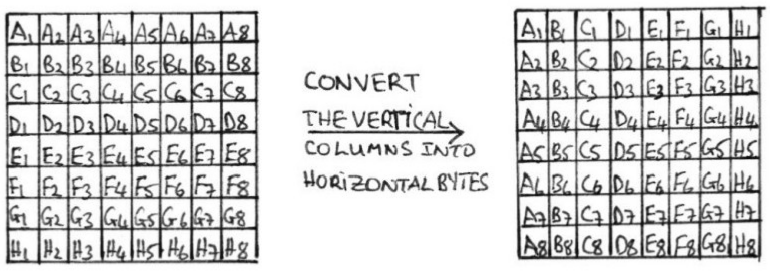


Figure 8-4: Transposition of a 8x8 alphanumeric character.

8.4.2 Printer Density Modes

The computer therefore takes the screen bytes and rotates them into the Epson vertical byte format. This data is then sent to the printer. The printer ,when in bit image mode,takes the 8 vertical bits and assigns them to 8 of the printer pin heads and depending on whether the bit is ON or OFF,determines the graphical pattern on the paper (where a 1=strike paper and 0=ignore) . Note that if the printer hasn't been set to bit image mode,the printer will take each data byte and print the ASCII character corresponding to it.

Thus,before a hardcopy of a screen dump is possible,we must make sure that the printer has been initialised to the bit imgae mode. There are 3 bit image modes available on my Epson compatible printer:

1. Single Density (480 vertical sequences or dots wide).
2. Double Density (960 dots wide).
3. Quadruple Density (1920 dots wide).

As you can see,as the density increases,so does the resolution or detail of the the hardcopy. What is actually happening is the size of the dot is getting smaller. Increased print density is useful if you have a computer generated painting which is bigger than the MTX screen,you can with the correct software join screens together in this manner and by selecting double density get both screens side by side.

To select which of these print modes is required is a simple task even from BASIC:

```
                      LPRINT CHR$(27);"K";CHR$(n);CHR$(m)
```

where n = #00 to #FF (LSB) when m = 0 (MSB)
and n = #00 to #E0 when m = 1.

Please refer to the printer manual,as to how these relate to the above print densities. Listing 8-4 provides a screen dump utility and is comprehensively documentated. As a task,extract the transposition subroutine and draw its flowchart. Refer to "MTX SCREEN DUMP" article ,see earlier.

Listing 8-4: High Resolution Graphic Screen Hardcopy,in single density print mode. NB: change the BITIMODE to DB 27,"L",0,1 & see the result.

```
10 put the graphic screen generating code here.
.....999

2600 CODE

GRAPHDUMP: LD HL,GRAPHICLF        ; SET LINEFEED FOR BIT IMAGE
           LD C,3                 ; MODE.
           CALL SCTRLCODE         ;
           XOR A                  ; SET HL=0000 THE QUICK WAY
           LD H,A                 ; INITIALISE BYTE COUNTER.
           LD L,A                 ; 6144 SCREEN BYTES TO SEND
           LD (COUNTER),HL        ; TO THE PRINTER. EVERY BYTE
                                  ; IN THE SCREEN WILL BE READ
                                  ; AS 8x8 BLOCKS & TRANSPOSED
                                  ; TO THE VERTICAL FORMAT.
                                  ; COUNTER KEEPS TRACK OF THE
                                  ; NUMBER OF 8x8 PATTERNS
                                  ; THAT ARE CONVERTED TO THE
                                  ; 8x8 VERTICAL FORMAT.
```

```
            LD HL,(PGTBASE)         ; START OF THE GRAPHIC SCREEN
            CALL VDPREADSEL         ; DATA TABLE TO BE READ.

GLOOP:      LD HL,BITIMODE          ; SET PRINTER TO BIT IMAGE
            LD C,4                  ; MODE.
            CALL SCTRLCODE          ;

READPGT:    LD HL,DATABUFF          ; READ A 8x8 PATTERN OR SCRN
            LD B,8                  ; IMAGE INTO A BUFFER.
RD1:        IN A,(#01)              ; READ THE DATA A BYTE AT A
            LD (HL),A               ; TIME AND STORE IN BUFFER.
            INC HL                  ; MOVE POINTER FORWARD BY 1.
            DJNZ RD1                ; REPEAT UNTIL ALL 8 BYTES
                                    ; ARE STORED.

EPSON:      PUSH HL                 ; SAVE DATABUFF-1 POSITION.
            LD C,8                  ; NUMBER OF DATA BYTES.
EP1:        LD B,0                  ; INITIALISE THE REGISTER
                                    ; TO HOLD THE VERTICAL BYTE
            LD D,8                  ; 8 BITS IN A BYTE.
EP2:        OR #00                  ; ZERO FLAGS (CAN USE XOR A)
            DEC HL                  ; MOVE POINTER TO THE BOTTOM
            RLC (HL)                ; OF THE 8 DATA BUFFER. THEN
                                    ; ROTATE DATA LEFT BY 1 BIT.
            JR NC,ZERO              ; THIS BIT IS MOVED INTO THE
                                    ; CARRY FLAG (C).IF C=0 THEN
                                    ; SET VERTICAL BIT TO ZERO.
            SET 7,B                 ; ELSE C=1 AND BIT =1. WE ARE

ZERO:       OR #00                  ; SETTING BIT 7 EVERYTIME WE
            DEC D                   ; PASS HERE. AFTER EVERY SET
            JR Z,SEND               ; THE VERTICAL BIT IS SHIFTED
            RR B                    ; RIGHT. THIS WILL BE DONE 7
                                    ; TIMES UNTIL THE VERTICAL
                                    ; BYTE IS READY FOR PRINTING.
            JP EP2                  ; REPEAT UNTIL BYTE READY.
SEND: CALL  HARDCOPY                ; SEND VERTICAL DATA BYTE TO
                                    ; THE PRINTER FOR PRINTING.
            POP HL                  ; RESTORE DATABUFF-1. WE
                                    ; HAVE ROTATED 1 BIT OF EACH
                                    ; OF THE 8 BYTES TO MAKE UP
                                    ; ONE VERTICAL BYTE. SEE
                                    ; FIGURE 8-4 FOR EXPLANATION
            DEC C                   ; REPEAT THIS PROCESS UNTIL
            JR NZ,EP1               ; ALL 8 BYTES HAVE BEEN
                                    ; TRANSPOSED.

            LD HL,(COUNTER)         ; INCREASE THE 8x8 PATTERN
            INC HL                  ; COUNTER. THERE ARE 3x256
            LD (COUNTER),HL         ; BLOCKS OF 8x8 PATTERNS,
                                    ; OR 768 PATTERNS TO SEND
                                    ; TO THE PRINTER.

ENDOFROW:   LD A,L                  ; 32 PATTERNS PER LINE. IN
            AND #1F                 ; ACTUAL FACT WE ARE TRACKING
            JR NZ,READPGT           ; WHERE WE ARE ON THE SCREEN
                                    ; ACCORDING TO THE GRAPHIC
```

```
                                  ; TEXT COORDINATE SYSTEM OF
                                  ; 32x24 SCREEN LOCATIONS.

ENDOFLINE:  LD B,#10              ; SEND LINEFEED CODE.
            CALL HARDCOPY         ;

ENDOFPGT:   LD A,3                ; 3 BLOCKS OF 256 PATTERNS.
            CP H                  ;
            JR NZ,GLOOP           ; REPEAT UNTIL ALL 768 SENT
                                  ; OR ALL 24 LINES SENT.

RESETPRINTER:LD B,#1B             ; SEND ESC
            CALL HARDCOPY         ;
            LD B,#40              ; SEND 64 OR @.
            CALL HARDCOPY         ;

EXIT:       RET                   ;


; SUBROUTINES

HARDCOPY:                         ; SEE LISTING 8-1

CHBRK:                            ; SEE LISTING 8-1

SCTRLCODE:                        ; SEE LISTING 8-3

VDPREADSEL:                       ; SEE LISTING 6-3


; PROGRAM DATA  AND VARIABLES

GRAPHICLF:  DB 27,"A",8           ; SET TO 8/72" BETWEEN LINES
BITIMODE:   DB 27,"K",0,1         ; SET TO SINGLE DENSITY MODE.
PGTBASE:    DW #0000              ; START OF GII BIT MODE SCRN.
COUNTER:    DS 2                  ;
DATABUFF:   DS 8                  ; STORES THE 8x8 SCREEN
                                  ; PATTERN.

2610 RETURN
```

Save as:

```
        SAVE "GRAPHDUMPTXT"                 (tape users)
        DISC (USER) SAVE "GRAPHDUMP.TXT"    (disc users)
```

The above program will only run when you have drawn graphics or text on the Graphics screen. For example:

```
10 VS 4
........ code for drawing or printing on graphics screen.

100 GOSUB 2600

110 STOP
```

Finally,to test your skills,rewrite the above code only in a more modular format,ie as small subroutines.

9.0 The Memotech Keyboard and Joystick

9.1 Intoduction

Memotech,a small British company,first came to my attention with there add-on range of peripherals (keyboard, ram expansion) for the Sinclair ZX81. Like all of Sir Clive's microcomputers : ZX80 , ZX81 , ZX Spectrum , QL ;the lack of a proper keyboard was a major drawback and probably cost Sinclair Research Ltd thousands of potential customers ,especially in the business arena with the QL.

This type of criticism (pre-QL) ,obviously influenced the people at Memotech. As a result ,the Memotech MTX series was given a full 79 key professional quality keyboard. The keyboard was split up into three sections : (1) the QWERTY keypad ; (2) the FUNCTION keypad ; and (3) the CURSOR and NUMERIC dual function keypad. At this time,no other major home computer micro (except the very expensive IBM clones) were graced with such a keyboard,not even the BBC 'B', CBM 64 ,Atari or ZX/QL series.

The Memotech MTX series keyboard has been designed to act internally as a 8 x 10 matrix ,ie 8 rows of 10 keys (columns) per row. In actual fact this relates to a 80 key keyboard. However,there are only 77 keys assigned to the extended ASCII character set,with keys 78 & 79 corresponding to the two RESET keys,at either side of the space bar key. No 80th key was provided. Note that,the actual relationship between the actual keyboard that we the users type and the internal computer format will be covered later.

This chapter was included for completeness BUT because of outside constraints it lacks the polished and learning examples of the other chapters. However, you should have by now grasped the principles and techniques of getting the most from your hardware and how to build modular software. This knowledge is applicable to this chapter also.

9.2 Detecting a Keypress

The 8-bit Z80 CPU has upto 256 individual communication lines (2^8 = 256). On the Memotech MTX series, ports 5 and 6 were reserved for selecting the rows (0 - 7) and reading the keys (0 - 9) on the keyboard. The Z80 likes to handle data in 8-bit or 1-byte chunks. As already stated,the MTX series is mapped to provide 8 rows of 10 keys. All ready the row range is in the chunk size for the Z80 to handle. However, the 10 column keys per row will require two 8-bit chunks to be read per row or per scan.

9.2.1 Selecting the Sense Line

A row or sense line is selected by telling the Z80 CPU to "latch" onto port 5. As mentioned above,the row number or sense line is activated when the correct bit number corresponding to the row is selected, see table 9-1. On the MTX series ,this activation is achieved by setting the bit "low" or switching the bit "off" ,ie set to "0". Table 9-1 clearly demonstrates this principle and the relationship between the row number and the bit pattern.

Table 9-1: The relationship between the row number or sense line with the bit pattern data. This data is used to select the row to sense or scan for keypresses. This data is latched on to PORT 05 of the Z80 CPU.

Row Number	**Binary Pattern**	**The Latched Byte Data**	
		DECIMAL	**HEXADECIMAL**
01	1 1 1 1 1 1 1 0	254	#FE
02	1 1 1 1 1 1 0 1	253	#FD
03	1 1 1 1 1 0 1 1	251	#FB
04	1 1 1 1 0 1 1 1	247	#F7
05	1 1 1 0 1 1 1 1	239	#EF
06	1 1 0 1 1 1 1 1	223	#DF
07	1 0 1 1 1 1 1 1	191	#BF
08	0 1 1 1 1 1 1 1	127	#7F

The Z80 CPU communicates with the keyboard with the command:

```
OUT (port number),data
```

Port number #05 is used by the CPU for selecting the row (data) to scan. See listing 9-1 as to how this is acheived in Z80 assembly code.

Listing 9-1: How the Z80 CPU communicates with the KEYBOARD.

```
SELROW5:   LD A,#EF               ; select row 5 to scan
           OUT (#05),A            ;
           RET                    ;
```

However,life isn't as simple as that,because in there wisdom ,the MTX series designers decided,whether voluntarily or due to some other constraint,that row number 1 should be selected as 1111 1101 and row number 2 selected as 1111 1110. This would be coded as: LD A,#FD ; OUT (#05),A and LD A,#FE ; OUT (#05),A respectively. All other rows are as Table 9-1.

9.2.2 Scanning the Sense Line

Once the correct row or sense line has been selected,we can then scan this row for any keypresses. When a key or keys have been pressed then the bit(s) that corresponds to that key(s) will be set to "0". However,as there are 10 keys per row,we will need 2 bytes for the keypress information. Ports 5 and 6 of the Z80 CPU were reserved for reading the 2 sections of the row, see Table 9-2. Each key in the row has a corresponding bit assigned to it. This is essential for dual or Multiple keypresses on the same or spread over a number of rows. The keys are read from left to right. Table 9-3, is a break down of the internal keyboard format.

Table 9-2: This table describes the relationship between the key with the Z80 CPU binary or bit pattern.

Column	**Read Z80**	**Single k/press data returned from scan**		
number	**CPU PORT**	**Binary Pattern**	**Decimal**	**Hexadecimal**
01	05	1111 1110	254	#FE
02	05	1111 1101	253	#FD
03	05	1111 1011	251	#FB
04	05	1111 0111	247	#F7
05	05	1110 1111	239	#EF
06	05	1101 1111	223	#DF
07	05	1011 1111	191	#BF
08	05	0111 1111	127	#7F
09	06	0000 0010	002	#02
10	06	0000 0001	001	#01

Table 9-3: How the Keyboard is seen by the MTXOS. The key mapping is for the Unshifted keypresses. Tables 9-4 & 9-5 for the Shifted and Alpha Lock Mappings.

HEX	**FE**	**FD**	**FB**	**F7**	**EF**	**DF**	**BF**	**7F**	**02**	**01**
FD	ESC	2	4	6	8	0	^	EOL	BS	F5
FE	1	3	5	7	9	-	\	PAG	BRK	F1
FB	CTRL	w	r	y	i	p	[	↑	TAB	F2
F7	q	e	t	u	o	@	LF	←	DEL	F6
EF	A/LCK	s	f	h	k	;	]	→		F7
DF	a	d	g	j	l	:	RET	HOME		F3
BF	SHIFT	x	v	n	,	/	SH	↓		F8
7F	z	c	b	m	.	_	INS	CLS	SP	F4

Table 9-4: The keyboard mapping for Shifted keypresses.

HEX	**FE**	**FD**	**FB**	**F7**	**EF**	**DF**	**BF**	**7F**	**02**	**01**
FD	ESC	"	$	&	(	0	~	8	BS	F13
FE	!	#	%	'	)	=	|	7	9	F9
FB	CTRL	W	R	Y	I	P	{	5	4	F10
F7	Q	E	T	U	O	'	LF	1	6	F14
EF	A/LCK	S	F	H	K	+	}	3		F15
DF	A	D	G	J	L	*	RET	2		F11
BF	SHIFT	X	V	N	>	?	SH	.		F16
7F	Z	C	B	M	<	_	0	RET	SP	F12

Table 9-5: The Keyboard mapping for Alpha Lock ON keypresses.

HEX	**FE**	**FD**	**FB**	**F7**	**EF**	**DF**	**BF**	**7F**	**02**	**01**
FD	ESC	2	4	6	8	0	^	EOL	BS	F5
FE	1	3	5	7	9	-	\	PAG	BRK	F1
FB	CTRL	W	R	Y	I	P	[	↑	TAB	F2
F7	Q	E	T	U	O	@	LF	←	DEL	F6
EF	A/LCK	S	F	H	K	;	]	→		F7
DF	A	D	G	J	L	:	RET	HOME		F3
BF	SHIFT	X	V	N	,	/	SH	↓		F8
7F	Z	C	B	M	.	_	INS	CLS	SP	F4

The MTX keyboard that we see consists of 4 main rows of keys plus a row with the Space bar and the two RESET keys. These keys have been mapped onto an 8x10 internal matrix which as shown above in tables 9-3, 9-4 and 9-5. All the odd rows of the 8x10 matrix correspond to the odd keys on the MTX keyboard going from left to right. And all the even rows correspond to the even positioned keys. Simple eh!!!

The Z80 commands for scanning or reading the status of a keypress(es) on any of the 8 rows are:

```
(1)  IN (#05),A  ; this reads key columns 1-8 ,see Table 9-2.
(2)  IN (#06),A  ; this reads key columns 9-10,see Table 9-2.
```

As you will have grapsed the actual coding for testing for a keypress or keypresses is a little complicated. However, Listing 9-2 will describe the basics.

Listing 9-2: This is the skeleton listing for reading the 8 rows of keys. However, I haven't included any testing routines for determining which keys in a row have been pressed but only test to see which row the keypress occurred.

```
10 GOSUB 230
20 PRINT PEEK(ROW)  :REM INSERT THE Z80 ADDRESS FOR ROW AND
30 REM AS  RESULT  IS  IN DECIMAL COMPARE
40 REM WITH TABLE 9-2 FOR ACTUAL ROW.
50 STOP


230 CODE

TESTKEYPRES:PUSH DE              ; SAVE REGISTERS
            PUSH BC              ;
            PUSH HL              ;
            LD HL,ROWVALUES      ; ROWS 1 TO 8 SEQUENTIALLY
            LD B,8               ; YES THERE IS 8 ROWS.
SCANROWS:   LD A,(HL)            ; SELECT THE ROW TO READ
            OUT (#05),A          ;
```

```
           IN A,(#05)             ; GET STATUS OF COLS 1-8
           LD D,A                 ; TEST IT LATER.
           IN A,(#06),A           ; GET STATUS OF COLS 9-10
           LD E,A                 ; TEST LATER.
           LD A,#FF               ; IF 11111111 THEN NO KEY
           CP D                   ; PRESSED IN COLS 1-8.
           JR NZ,KEYPRESS         ; IF D<#FF THEN KEYPRESSED.
           LD A,3                 ; IF 00000011 THEN NO KEY
           CP E                   ; PRESSED IN COLS 9-10.
           JR NZ,KEYPRESS         ; ID E<3 THEN KEYPRESSED.
           INC HL                 ; POINT TO NEXT ROW.
           DJNZ SCANROWS          ; REPEAT FOR ALL 8 ROWS
                                  ; IF REQUIRED.
EXIT:      POP HL                 ; RESTORE REGISTERS
           POP BC                      ;
           POP DE                      ;
           RET                    ;

KEYPRESS:  LD A,(HL)              ; A=ROW THE THE KEY PRESS
           LD (ROW),A             ; OCCURRED,SAVE IT.
                                  ; INSERT HERE THE CODE FOR
                                  ; TESTING THE ROW KEY PATTERNS
                                  ; FOR WHICH KEY(S) WERE PRESSED

           JR EXIT                ;

ROWVALUES: DB #FD,#FE,#FB,#F7,#EF,#DF,#BF,#7F
ROW:       DS 1

240 RETURN
```

Save as:

```
           SAVE "ROWSCANTXT"                      (tape users)
           DISC (USER) SAVE "ROWSCAN.TXT"         (disc users)
```

Reload listing 9-2. Type RUN <RET>. When you press a key or multiple keys the routine will print the row which it detected the keypress first. Remember that,the program scans the rows sequentially from top to bottom.

9.2.3 Testing for a Keypress

As already stated,to check for a keypress,we must perform an IN from Ports 5 and 6 to get the status of keys 1-8 and 9-10 respectively,ie READ the status. Also,the status of each key on the scanned row is determined by the value of the corresponding bit in the Port 5 & 6 data bytes,see last section,ie if bit = 0 then keypress and if bit = 1 then no keypress.

Unlike MTX BASIC,the Z80 assembler has access to logical operator commands: AND/OR/XOR which allow the user to test individual bits in a byte.The Z80 programmer can also use the commands: BIT/SET/RES, which work on specific bits of information. Refer to a good Z80 book for details of these commands.

The use of the BIT/SET and RES Z80 commands to test for keypresses,is only useful if you are testing for a particular bit or key ,ie testing to see if <BRK> has been pressed. In actual fact,we have already covered the test to see if <BRK> has been pressed in chapter 8 ,see listing 8-1 ,subroutine CHBRK.

Finally,an essential part of any keyboard scanning routine,is that test for no keypress on a particular row. By scanning the row in question and testing the row data against the no keypressed bit pattern,we can save a lot of processor time than if we had to test each bit individually. Listing 9-2,has already covered this,but I think it is important enough for a reminder . The no keypress bit pattern for PORT 5 is 11111111 and for PORT 6 is 00000011.

9.3 Reading the Joystick Ports

When the Memotech MTX series was being developed a great number of computer journalists and the public were screaming out for a good Z80 machine with a built-in Z80 assembler,advanced sound and graphics,joystick ports,proper keyboard ,and buisness capabilities. When the MTX was released,it was claimed to be technical excellent,offering as Jack tramiel of Atari would say,"power without the price". However,the public never took to it and the MTX since then has had to take a back seat in British computing.

The inclusion of two joystick ports was meant to attract the games enthausist and the games programmer,but it never did. However,a number of excellent joystick games have been written for the MTX series. The ability to use a joystick,not only saves the keyboard from wear and tear but it allows better arcade games control. Positioned at the rear of the MTX keyboard and to the left they are two ATARI D-type joystick ports. This section describes how to write a program for testing the joystick ports.

Reading the joystick is no different from reading the keyboard. This is because the MTX designers,mapped both the joystick ports to mimic specific keys on the keyboard. This is particularly important to the programmer as one set of coding accommodates both keyboard and joystick games players. The right hand joystick keys are mapped onto the up,down,right,left cursor and Home keys of the keyboard. Why don't you figure out which keys are mapped onto the LEFT Hand joystick port. Listing 9-3 ,demonstrates how the right hand joystick is programmed and specific keys tested for.

Listing 9-3: A set of subroutines for scanning the Right Hand Joystick port. These subroutines can be easily implemented into your own listings.

```
250 CODE

; SUBROUTINES

RHJOYSTICK:NOP                        ; SEE TABLE 9-3,IN THIS YOU WILL
                                      ; NOTICE THAT ALL THE RHJOYSTICK
                                      ; KEYS ARE ALL ON THE SAME COLUMN.
                                      ; THIS MAKES PROGRAMMING SIMPLER.

FIREKEY:    LD A,#DF                  ; TESTS FOR HOME (KBD) OR FIRE
            CALL SCANCOL8             ; (JOY) BEING PRESSED.
            CALL Z,FIRE               ; IF SELECTED THEN JUMP TO THE
                                      ; FIRE SUBROUTINE.

LEFTCUR:    LD A,#F7                  ; TEST FOR LEFT (KBD OR JOY) BEING
            CALL SCANCOL8             ; PRESSED. IF PRESSED THEN GOTO
            CALL Z,LEFT               ; LEFT SUBROUTINE.

RIGHTCUR:   LD A,#EF                  ; TEST FOR THE RIGHT (KBD OR JOY)
            CALL SCANCOL8             ; ETC.
            CALL Z,RIGHT              ;

UPCUR:      LD A,#FB                  ; TEST FOR UP (KBD OR JOY). ETC.
            CALL SCANCOL8             ;
            CALL Z,UP                 ;

DOWNCUR:    LD A,#BF                  ; TEST FOR DOWN (KBD OR JOY). ETC.
            CALL SCANCOL8             ;
            CALL Z,DOWN               ;

            CALL LIVELOST             ; SUBROUTINE TO TEST FOR A LIVE
                                      ; LOST IN AN ARCADE GAME.

REPEAT:     JR RHJOYSTICK             ;

SCANCOL8:   OUT (#05),A               ; SELECT ROW TO SCAN.
            IN A,(#05)                ; READ SELECTED ROW.
            CP #7F                    ; IS KEY ON COL 8 BEEN PRESSED.
                                      ; IF IT HAS,Z-FLAG IS SET.
            RET                       ;

FIRE:                                 ; CHECK BULLETS LEFT
                                      ; IF SOME LEFT FIRE.
                                      ; TEST IF BULLET HIT.
                                      ; IF HIT :
                                      ; UPDATE SCREEN AND SCORE.
                                      ; HAVE YOU WON,ETC.
            RET                       ;

DOWN:                                 ; CHECK SCREEN BOUNDRY.
                                      ; IF OFF BOTTOM,IS SCROLLING
                                      ; POSSIBLE,AS IN A W/PROCESSOR.
                                      ; ETC.
            RET                       ;

LEFT:       RET                       ; AS FOR DOWN
```

```
UP:         RET                    ; AS FOR DOWN.

RIGHT:      RET                    ; AS FOR DOWN.

LIVESLOST:  RET                    ; INSERT CODE HERE IF REQUIRED.


260 RETURN
```

Save as:

```
            SAVE "RHJOYSTTXT"                (tape users)
            DISC (USER) SAVE "RHJOYST.TXT"   (disc users)
```

9.4 MTX Series ROM BIOS - Keyboard Routines

You may be wondering,why bother writing our own keyboard specific routines which address the hardware directly rather than use the built-in MTX Series keyboard routines stored on ROM. For example,the above listing (9-3) could have been written as:

Listing 9-4: Listing 9-3 redone to use the ROM keyboard code.

```
270 CODE

RHJOY:      XOR A                  ; RESET FLAGS
            CALL #0079             ; ACCESS ROM KEYBOARD CODE.
            JR Z,RHJOY             ;

TESTJOY:    CP 26                  ; IS FIRE PRESSED?
            CALL Z,FIRE            ;
            CP 8                   ; IS LEFT PRESSED?
            CALL Z,LEFT            ;
            CP 10                  ; IS DOWN PRESSED?
            CALL Z,DOWN            ;
            CP 11                  ; IS UP PRESSED?
            CALL Z,UP              ;
            CP 25                  ; IS RIGHT PRESSED?
            CALL Z,RIGHT           ;
            CALL LIVESLOST         ;
            JR RHJOY               ;

; SUBROUTINES FIRE/LEFT/DOWN/UP/LIVESLOST AS FOR LISTING 9-3.

280 RETURN
```

This method is actually very useful,if you are writing wordprocessor; spreedsheet type applications. Also,because most systems use the ASCII system for key recognition,adapting listing 9-4 to other machines like the MSX is a simple task of replacing the ROM keyboard call to the one implemented on the MSX. However,if you are writing fast interactive arcade games, where speed of movement and reaction times are crucial ,then we will obviously have to write hardware specific routines, as in Listing 9-3.

Another advantage for writing your own OS utilities for the MTX series,in particular,is because you can write your software so that the code will run on either the MTX OS or on the CPM enviroment. Both Operating Systems ,read the keyboard differently but both read the hardware in the same way. With games software this ability to write one program for either system is a big saving in program development. Throughout this book,I have been building up your library of specific hardware orientated software,so that you can get the most out of your code and to see how to go about writing an operating system for instance.

The Memotech MTX series ROM code is accessed through a gateway at #0079. This instructs the OS to RUN the code on ROM PAGE 0 at #3618. The technique for Page switching was covered in chapter 1. The example chosen is relevant to this section. Therefore go back and re-read that subsection. The CALL #0079, gateway returns the ASCII value of the key pressed. The ASCII value is returned in Register A. If no key was pressed then the Z-flag would be set. A simple wait for keypress routine would be.

Listing 9-5: Wait for a keypress using the ROM keyboard BIOS.

```
290 CODE

ROMKEYS:   XOR A                  ; RESET THE Z-FLAG AND CLEAR A.
           CALL #0079             ; KEYBOARD SCAN GATEWAY.
           JR Z,KEYS              ; IF Z SET THEN NO KEY PRESSED.
           RET                    ; RETURN WHEN KEY IS PRESSED.

300 RETURN
```

In actual fact,the CALL #0079 subroutine returns three parameters in the MTX system variables table, starting at #FD7B:

```
LASTKEY:   DS 1                   ; HOLDS AN INTERNAL FORMAT OF KEY
                                  ; PRESSED.
ROWSCAN:   DS 1                   ; HOLDS THE DRIVE LINE OR ROW No.
LASTASC:   DS 1                   ; HOLDS THE ASCII KEYPRESS NUMBER.
```

9.5 Operating System Independent Keyboard Utility Routines

To conclude this chapter,I have included my OS independent full keyboard scan routine,see Listing 9-6. This code has been designed to be directly interchangeable with the MTX ROM CALL #0079. For example,again the ASCII keypress result is held in register A on returning from KEYS. The Z-flag is set if no key is pressed. Also ROWSCAN and LASTASC are also returned,for completeness. Finally,I have included the option of changing the delay between keypresses,so as to avoid key repetition occuring. This delay can have a value of 0 (fast) to 255 (slow). It is set by passing the value in register A to the keys subroutine,see listing 9-6.

To compliment the code in Listing 9-6,I have included the original flow diagram that I used when I was designing this listing. As a task,why not produce a similar flow diagram for the finished listing. For flow diagram symbols, refer to your MTX Operators Manual, technical section.

Listing 9-6: OS independent keyboard scan routine.

```
10 CODE

RAMKEYS:   XOR A                  ; EQUIVALENT TO LISTING 9-5.
           LD A,8                 ; KEYBOUNCE DELAY
           CALL KEYS              ; INSERT THE ADDRESS FOR KEYS
           JR Z,RAMKEYS           ;
           RET                    ;


20 LET A=PEEK(ASCIIKEY)
30 CSR 10,10:PRINT A
40 IF A<332 OR A>127 THEN GOTO 10
50 CSR 10,14: PRINT CHR$(A)
60 GOTO 10
70 STOP


2700 CODE

KEYS:      CP  0                  ; IF A=0 ON ENTRY THEN USE
           JR Z,DEFWAIT           ; DEFAULT KEYBOUNCE VALUE.
           LD (KEYBOUNCE),A       ; ELSE USE THE USER ONE.

DEFWAIT:   DI                     ; DISABLE THE INTERRUPTS TO
           EXX                    ; AVOID CLASHING WITH THE
           PUSH IX                ; BUILT-IN KEYBOARD ROUTINES
           CALL TESTMODE          ; NOW SCAN FOR KEYPRESSES.
           EXX                    ; REMEMBER TO RESTORE REG's
           POP IX                 ; ON LEAVING KEYS & ALSO TO
           EI                     ; ENABLE THE BASIC INTERRUPT
           EX AF,AF'              ; NOW WAIT A FEW MOMENTS
           LD A,(KEYBOUNCE)       ; BETWEEN KEYPRESSES,SO THAT
           LD B,A                 ; WE DON'T END UP WITH THE
                                  ; KEYPRESS REPEATING x TIMES
WAITLOOP:  HALT                   ; ACROSS THE SCREEN.
           DJNZ WAITLOOP          ;
           EX AF,AF'              ;
           AND A                  ; LASTLY,CHECK TO SEE IF A
           CP 0                   ; KEY WAS PRESSED. IF NOT
           RET                    ; THEN Z-FLAG SET. EXIT.

TESTMODE:  LD B,3                 ; FIRSTLY,TEST TO SEE IF
           LD HL,ROWBYTE          ; KEYS CTRL/SHIFT/ESC HAVE
SR5LOOP:   LD A,(HL)              ; BEEN PRESSED. IF SET THEN
           LD C,A                 ; SET APPROPRIATE FLAG. NOW
           EX AF,AF'              ; TEST ALL ROWS TO SEE IF
           LD A,C                 ; ANOTHER KEY IS PRESSED AT
```

```
            OUT (5),A               ; THE SAME TIME,ie <CTRL> P
            IN A,(5)                ; IN CPM SENDS THE SCREEN
            LD D,A                  ; OUTPUT TO THE PRINTER. IF
            IN A,(6)                ; NO OTHER KEY PRESSES WITH
            LD E,A                  ; IT,THEN EXIT.
            LD A,C                  ; IF ESC/CTRL/SHIFT HAD NOT
            LD C,1                  ; BEEN PRESSED THEN SCAN ALL
            AND A                   ; ROWS FOR A KEYPRESS. IF NO
            CP #BF                  ; KEYPRESS THEN EXIT.
            JR NZ,SR5OVR            ; REMEMBER,WHEN TESTING FOR
            LD C,65                 ; SHIFT,IT HAS TWO KEY COLS.
SR5OVR:     LD A,D                  ;
            XOR #FF                 ;
            AND C                   ;
            CP 0                    ;
            JR NZ,EXITSR5           ;
            INC HL                  ;
            DJNZ SR5LOOP            ;

SINGKEY:    LD BC,UNSHROWTAB        ; IF SINGLE KEYPRESS ONLY
            LD (ROWTABLE),BC        ; THEN LOAD THE UNSHIFTED
            JR TESTROWS             ; KEY ASCII TABLE.

EXITSR5:    LD A,2                  ; IF SHIFT/CTRL OR ESC IS
            AND A                   ; PRESSED THEN JUMP TO THE
            CP B                    ; CORRECT SUBFUNCTION & LOAD
            JR C,SHIFT              ; THE CORRECT ASCII TABLE AS
            JR Z,CTRL               ; DONE IN SINGKEY.
            JR ESC                  ;

SHIFT:      LD HL,SHTABLE           ; LOAD THE SHIFTED ASCII
            SET 6,D                 ; TABLE.
            LD BC,SHROWTAB          ;
            JR SCECODE              ;

CTRL:       LD HL,CTTABLE           ; LOAD THE CTRLED ASCII
            LD BC,CTROWTAB          ; TABLE.
            JR SCECODE              ;


ESC:        LD A,#1B                ; NO TABLE TO LOAD. BUT ONE
            RET                     ; COULD BE ADDED. EXIT TO
                                    ; BASIC.


SCECODE:    LD A,1                  ; THIS SUBROUTINE SETS THE
            LD (MODEFLAG),A         ; MODE FLAG. MODE REFERS TO
            LD (ROWTABLE),BC        ; CTRL OR SHIFT. THE CODE
            SET 0,D                 ; THEN CHECKS FOR KEYPRESSES
            PUSH DE                 ; ON THE SAME ROW AS SHIFT
            POP BC                  ; OR CTRL. IF NO OTHER KEY
            EX DE,HL                ; IS PRESSED ON THESE ROWS.
            AND A                   ; THEN THE OTHER ROWS ARE
            LD HL,#FF03             ; TESTED USING THE TESTROWS
            SBC HL,BC               ; SUBROUTINE.
            JR Z,TESTROWS           ;
            PUSH HL                 ; IF ANOTHER KEY ON THE SAME
```

```
MATCHOK:    XOR A                   ; ROW AS SHIFT OR CTRL IS
            LD (MODEFLAG),A         ; PRESSED THEN RESET THE
            EX DE,HL                ; MODE FLAG AND GET THE
            JR GETASCII             ; ASCII VALUE OF THE DUAL
                                    ; KEYPRESS.

TESTROWS:   LD IX,(ROWTABLE)        ; THIS SUBROUTINE SIMPLY
ROWLOOP:    LD A,(IX+0)             ; TESTS THE ROWS OF THE
            CALL SCANROW            ; KEYBOARD MATRIX FOR KEY-
            LD A,(IX+0)             ; PRESSES. THE ROWS TO BE
            INC IX                  ; TESTED ARE FOUND AT THE
            AND A                   ; ADDRESS POINTED TO BY
            CP #FF                  ; ROWTABLE. THEREFORE,IT IS
            JR NZ,ROWLOOP           ; POSSIBLE TO TEST ANY ROW
            LD A,(MODEFLAG)         ; IN ANY ORDER.
            AND A                   ;
            CP 0                    ;
            JR Z,RETKEYS            ;
            XOR A                   ;
            LD (MODEFLAG),A         ;
RETKEYS:    RET                     ;

SCANROW:    LD C,A                  ; THIS IS THE WORKHORSE OF
            EX AF,AF'               ; SUBROUTINE TESTROWS. IT
            LD A,C                  ; TELLS THE HARDWARE TO
            OUT (5),A               ; SELECT ROWS AND TO SCAN OR
            IN A,(5)                ; READ THEM FOR KEYPRESS(ES).
            LD B,A                  ;
            IN A,(6)                ;
            LD C,A                  ;
            AND A                   ;
            LD HL,#FF03             ;
            SBC HL,BC               ;
            RET Z                   ;

ELSEMATCH:  LD A,(MODEFLAG)         ; IF KEY PRESSED THEN CHECK
            AND A                   ; TO SEE IF IT IS A DUAL KEY
            CP 1                    ; PRESS,ie SHIFT <f1>.
            JR Z,MATCHOK            ;

ELSECAPS:   EX AF,AF'               ; IF THE ALPHA (CAPS) LOCK
            PUSH AF                 ; KEY BEEN PRESSED THEN
            EX AF,AF'               ; GOTO UPDATCAPS.
            POP AF                  ; ELSE CHECK TO SEE IF IT
            AND A                   ; IS ANOTHER KEY BEEN
            CP #EF                  ; PRESSED.
            JR NZ,ELSENOCH          ;
            BIT 0,B                 ;
            JR NZ,ELSENOCH          ;
UPDATCAPS:  LD A,(CAPSFLAG)         ; LOAD THE CAPS FLAG AND
            XOR 1                   ; TOGGLE IT ON OR OFF FROM
            LD (CAPSFLAG),A         ; ITS PREVIOUSLY STORED VALUE
            LD A,1                  ; . NOW EXIT TO BASIC.
            POP DE                  ;
            RET                     ;
```

```
ELSENOCH:   LD A,(CAPSFLAG)         ; HAS THE CAPS FLAG BEEN SET
            AND A                   ; PREVIOUSLY. IF SO THEN LOAD
            CP 0                    ; CAPS ASCII TABLE. IF NOT
            JR Z,ELSENOSH           ; THEN IT MUST BE A UNSHIFTED
            LD HL,CAPSTABLE         ; KEYPRESS.
            JR GETASCII             ;

ELSENOSH:   LD HL,UNSHTABLE         ; LOAD THE UNSHIFTED ASCII
                                    ; TABLE.

GETASCII:   EX AF,AF'               ; THIS SUBROUTINE CONVERTS
            LD (ROWSCAN),A          ; THE KEYPRESS(ES) INTO THE
            AND A                   ; CORRESPONDING ASCII KEY
            LD D,10                 ; VALUE,ACCORDING TO THE
            PUSH BC                 ; THE LOADED ASCII TABLE.
            CALL GETKEYPOS          ; THIS VALUE IS RETURNED TO
            POP BC                  ; RAMKEYS IN LINE 10, AND IS
            LD A,3                  ; ALSO STORED IN ASCIIKEY.
            CP C                    ; THE ROW IS ALSO SAVED IN
            JR NZ,P6MATCH           ; RAM AT ROWSCAN.

ELSEP5:     LD A,B                  ;
            LD D,1                  ;
            CALL GETKEYPOS          ;
            JR ASCIIVAL             ;

P6MATCH:    LD A,C                  ; THIS SECTION IS USED FOR
            ADD A,L                 ; KEYPRESSES IN COLS 9 & 10.
            ADD A,7                 ;
            LD L,A                  ;
ASCIIVAL:   LD A,(HL)               ;
            LD (ASCIIKEY),A         ;
            POP DE                  ;
            RET                     ;

GETKEYPOS:  XOR 255                 ; THIS CODE PINPOINTS THE
            LD BC,#0701             ; ASCII COL AND ROW IN THE
            LD E,0                  ; LOADED TABLE,SO THAT THE
POSLOOP:    CP C                    ; KEYPRESS ASCII VALUE CAN
            JR Z,EXITLOOP           ; BE EXTRACTED.
            SLA C                   ;
            EX AF,AF'               ;
            LD A,E                  ;
            ADD A,D                 ;
            LD E,A                  ;
            EX AF,AF'               ;
            AND A                   ;
            DJNZ POSLOOP            ;
EXITLOOP:   LD A,L                  ;
            ADD A,E                 ;
            LD L,A                  ;
            RET                     ;

KEYBOUNCE:  DB #00                  ;
ROWSCAN:    DS 1                    ;
ASCIIKEY:   DS 1                    ;
```

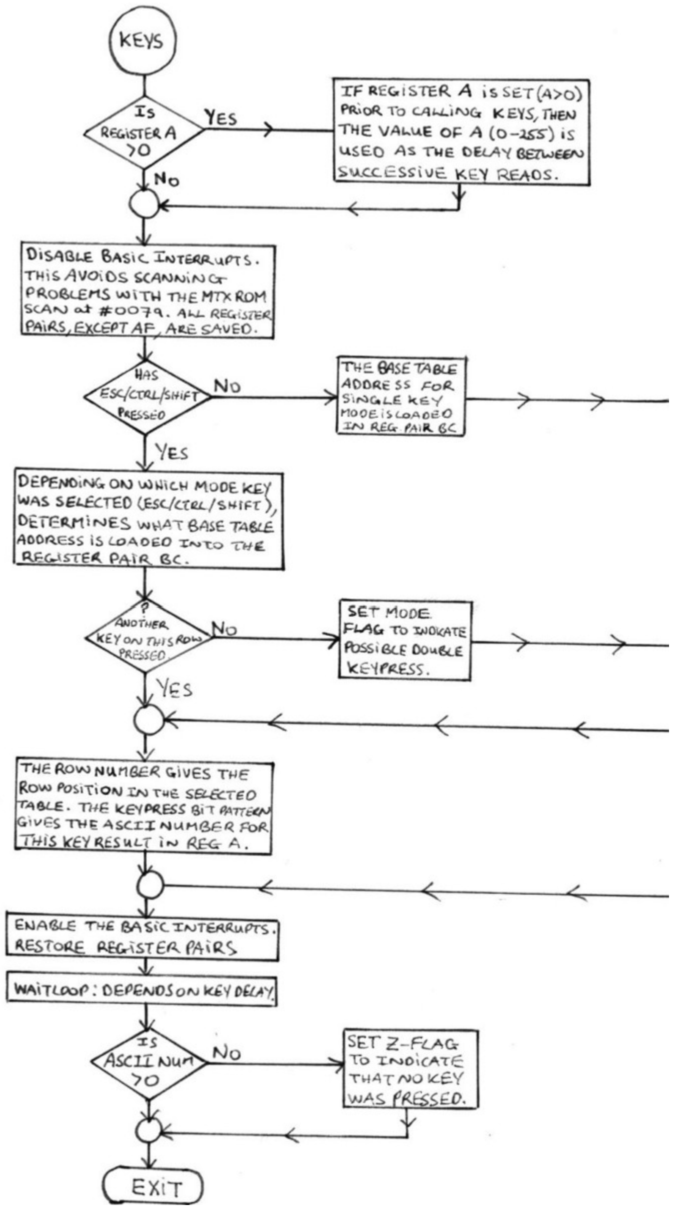
```
UNSHTABLE: DB #31,#33,#35,#37,#39,#2D,#5C,#1D,#80,#03
           DB #1B,#32,#34,#36,#38,#30,#5E,#05,#84,#08
           DB #00,#77,#72,#79,#69,#70,#5B,#0B,#81,#09
           DB #71,#65,#74,#75,#6F,#40,#0A,#08,#85,#7F
           DB #00,#73,#66,#68,#6B,#3B,#5D,#19,#86,#00
           DB #61,#64,#67,#6A,#6C,#3A,#0D,#1A,#82,#00
           DB #00,#78,#76,#6E,#2C,#2F,#00,#0A,#87,#00
           DB #7A,#63,#62,#6D,#2E,#5F,#15,#0C,#83,#20

SHTABLE:   DB #21,#23,#25,#27,#29,#3D,#7C,#37,#88,#39
           DB #1B,#22,#24,#26,#28,#30,#7E,#38,#8C,#08
           DB #00,#57,#52,#59,#49,#50,#7B,#35,#89,#34
           DB #51,#45,#54,#55,#4F,#60,#0A,#31,#8D,#36
           DB #00,#53,#46,#48,#4B,#2B,#7D,#33,#8E,#00
           DB #41,#44,#47,#4A,#4C,#2A,#0D,#32,#8A,#00
           DB #00,#58,#56,#4E,#3C,#3F,#00,#2E,#8F,#00
           DB #5A,#43,#42,#4D,#3E,#5F,#30,#0D,#8B,#20

CTTABLE:   DB #11,#13,#15,#17,#19,#0D,#1C,#1D,#01,#03
           DB #1B,#12,#14,#16,#18,#10,#1E,#05,#05,#08
           DB #00,#17,#12,#19,#09,#10,#1B,#0B,#02,#09
           DB #11,#05,#14,#15,#0F,#00,#0A,#08,#06,#1F
           DB #00,#13,#06,#08,#0B,#1B,#1D,#19,#07,#00
           DB #01,#04,#07,#0A,#0C,#1C,#0D,#1A,#03,#00
           DB #00,#18,#16,#0E,#0C,#0F,#00,#0A,#08,#00
           DB #1A,#03,#02,#0D,#0E,#1F,#15,#0C,#04,#20

CAPSTABLE: DB #31,#33,#35,#37,#39,#2D,#5C,#1D,#80,#03
           DB #1B,#32,#34,#36,#38,#30,#5E,#05,#84,#08
           DB #00,#57,#52,#59,#49,#50,#5B,#0B,#81,#09
           DB #51,#45,#54,#55,#4F,#40,#0A,#08,#85,#7F
           DB #00,#53,#46,#48,#4B,#3B,#5D,#19,#86,#00
           DB #41,#44,#47,#4A,#4C,#3A,#0D,#1A,#82,#00
           DB #00,#58,#56,#4E,#2C,#2F,#00,#0A,#87,#00
           DB #5A,#43,#42,#4D,#2E,#5F,#15,#0C,#83,#20

ROWBYTE:   DB #BF,#FB,#FD,#FF
CAPSFLAG:  DB 0
MODEFLAG:  DB 0
ROWTABLE:  DS 2

UNSHROWTAB:DB #FE,#FD,#FB,#F7,#EF,#DF,#BF,#7F,#FF
SHROWTAB:  DB #FE,#FD,#FB,#F7,#EF,#DF,#7F,#FF
CTROWTAB:  DB #FE,#FD,#F7,#EF,#DF,#BF,#7F,#FF
ESROWTAB:  DB #FE,#FB,#F7,#EF,#DF,#BF,#7F,#FF

2710 RETURN
```

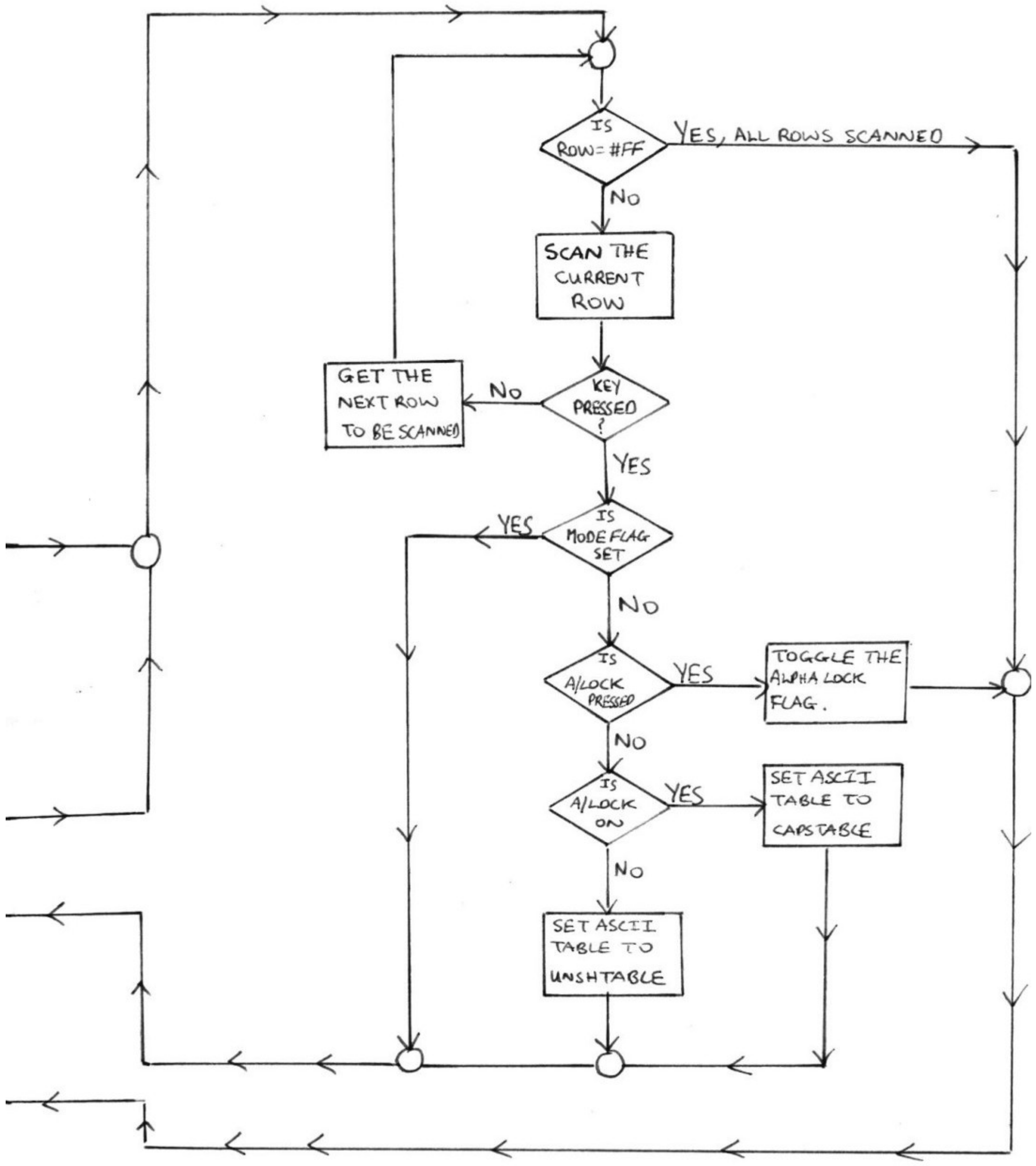
Save as:

```
           SAVE "RAMKBDTXT"                  (tape users)
           DISC (USER) SAVE "RAMKBD.TXT"     (disc users)
```

Reload this listing. Type RUN <RET>. Whenever,you press a key,its ASCII value will be displayed on the screen. If,the ASCII value is between 32 and 127 ,then its corresponding character pattern will also be displayed.

Figure 9-1: Original Flow Diagram ,from which the KEYS Subroutine was Developed from.


KEYS
IS REGISTER A >0
YES
IF REGISTER A IS SET (A>0) PRIOR TO CALLING KEYS, THEN THE VALUE OF A (0-255) IS USED AS THE DELAY BETWEEN SUCCESSIVE KEY READS.
NO
DISABLE BASIC INTERRUPTS. THIS AVOIDS SCANNING PROBLEMS WITH THE MTX ROM SCAN at #0079. ALL REGISTER PAIRS, EXCEPT AF, ARE SAVED.
HAS ESC/CTRL/SHIFT PRESSED
NO
THE BASE TABLE ADDRESS FOR SINGLE KEY MODE IS LOADED IN REG PAIR BC
YES
DEPENDING ON WHICH MODE KEY WAS SELECTED (ESC/CTRL/SHIFT), DETERMINES WHAT BASE TABLE ADDRESS IS LOADED INTO THE REGISTER PAIR BC.
? ANOTHER KEY ON THIS ROW PRESSED
NO
SET MODE FLAG TO INDICATE POSSIBLE DOUBLE KEYPRESS.
YES
THE ROW NUMBER GIVES THE ROW POSITION IN THE SELECTED TABLE. THE KEYPRESS BIT PATTERN GIVES THE ASCII NUMBER FOR THIS KEY RESULT IN REG A.
ENABLE THE BASIC INTERRUPTS. RESTORE REGISTER PAIRS
WAITLOOP: DEPENDS ON KEY DELAY
IS ASCII NUM >0
NO
SET Z-FLAG TO INDICATE THAT NO KEY WAS PRESSED.
EXIT



IS ROW = #FF
YES, ALL ROWS SCANNED
NO
SCAN THE CURRENT ROW
GET THE NEXT ROW TO BE SCANNED
NO
KEY PRESSED ?
YES
YES
IS MODE FLAG SET
NO
IS A/LOCK PRESSED
YES
TOGGLE THE ALPHA LOCK FLAG.
NO
IS A/LOCK ON
YES
SET ASCII TABLE TO CAPSTABLE
NO
SET ASCII TABLE TO UNSHTABLE

10.0 MTX SOUND

10.1 Introduction

In a recent PCWeekly survey,9 out of 10 people expect their computer to be able to make reasonable sound,with two-thirds of these expecting at least three channel sound.

The two main programmable sound generator (PSG) integrated chips used in today’s modern home computers are General Instruments AY-8910 and the Texas Instruments SN 76489A. The AY-8910 is the most commonly used PSG as it has stereo output over 8 octaves and is used in the following micros: MSX, Amstrad CPC, Einstein range, Spectrum 128 & plus 2 and in the Atari ST range. The SN 76489A is less common but is used on two of the more sophisticated micros,the powerful 6502 cpu BBC micro and on the Z80 cpu Memotech MTX series. The SN 76489A is less powerful as only 4 octaves of mono sound can be generated.

The SN 76489A PSG can produce three seperate voices and one noise channel,thus allowing harmonies to be created. As the MTX series has no onboard speaker,the sound is directed through the TV/monitor speaker or through the standard HiFi socket at the rear of the MTX. The latter has the advantage of giving quality sound output and is very handy for recording any masterpieces composed.

Unfortunately for MTX owners there are only two composer/music programs. The main objective of this chapter is to take the lid off the PSG. As yet nobody has tried to explain the workings of the PSG or how to program it from within the built in assembler. I hope this chapter opens up new avenues for the MTX programmer and inspire somebody to write an excellent composer program.

10.2 Description of the PSG

The Texas Instruments SN 76489A IC is a bipolar IC and is capable of producing complex sound generation. The device consists of three programmable tone generators,a programmable noise generator,a clock scaler,individual generator attenuators (ie,volume controls) and an audio summer output buffer. The PSG has a parallel 8-bit interface through which the microprocessor transfers the data which controls the audio output. A pin out or top view of the PSG is given in figure 10-1.

The PSG has a variety of internal registers,0-7 (ie R0-R7),which are used to control the activities of each of the three voices. Registers 1,3 and 5 are used to control the volume of the three tone channels and register 7 for the noise channel. Registers 0,2 and 4 are used to produce a square wave signal of varying frequency,see figure 10-2.

A seperate noise generator,register 6,provides a more random waveform. The audio scanner output buffer mixes the outputs of the frequency generators with the volume and noise signals to produce the eventual sound we here from the speaker.

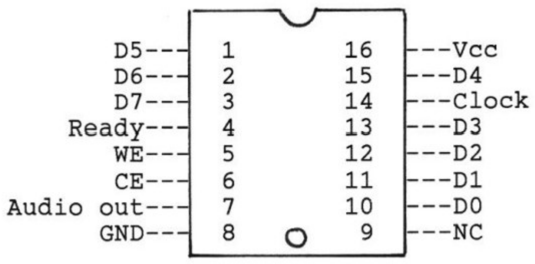


Figure 10-1: Top view of the TI SN 76489A PSG

Figure 10-2: Two square waves of differing frequency

10.3 CPU - PSG Communication

The microprocessor,Z80,interfaces with the PSG by means of 8 data lines and 3 control lines,WE,CE and Ready. The Z80 selects the PSG by placing CE into the true state (low voltage). Unless CE is true,the WE signal strobes the contents of the data bus to the appropriate control register,as the PSG has a parallel 8-bit interface to the Z80. The data bus contents must be valid at this stage.

If the last paragraph was a bit confusing then here it is again. To write data to the PSG,the Z80 sends valid data via 8 data lines D0-D7 to output port 6 on the Z80. The data waits here until a dummy read to port 3,sends this data to the PSG,see listing 10-1.

Listing 10-1: CPU - PSG communication.

```
Cputopsg:  OUT (6),A              ; load register A with data and
                                  ; store at port 6
           IN A,(3)               ; This dummy read sends the data
                                  ; from the Z80 to the PSG.
           RET                    ; Return to calling routine.
```

There is one point to beware of and that is that 32 clock cycles or T-states must elapse before another dummy read can be performed. More information on this can be found on page 243 of the MTX manual, Technical section.

10.4 Volume Control

Three tone generators,0-2,are available in MTX basic. In order to create music we must specify the music using the following MTX basic command:

```
            SOUND  c,f,v
```

```
where       c = tone generator or sound channel,range 0-2
            f = frequency of the note,range 10-1020,*
            v = volume of the note,range 0 (min) to 15 (max)
```

*,note that this is a pseudo frequency range,the actual frequency range is 12500 Hz to 122 Hz repectively. The actual frequency is calculated from equation (1) section 10.5
. A list of pseudo frequencies and their equivalent actual frequencies are given on page 185 to 187 of the MTX manual.

Unfortunately the MTX machine code programmer cannot use this format as the PSG is configured differently and the extra code required to mimic this command is lengthier and takes longer to execute. The PSG controls the volume of the three available tone generators via three dedicated volume registers 1,3 and 5. The PSG requires only one byte of information to select the register and the volume to be outputted. The upper nibble or upper 4-bits are used define the register and the lower nibble defines the volume of the note. I emphasise at this point that the volume range used by the PSG is the reverse of basic,ie 0 (max) and 15 is minimum.
The upper nibble patterns give the volume register,see table 10-1.

Table 10-1: Volume register select

Sound channel	**PSG register**	**Upper Nibble pattern**	**Data to select PSG register**
0	1	0 0 0 1	16
1	3	0 0 1 1	48
2	5	0 1 0 1	80
3*	7	0 1 1 1	112

* =   This is the noise volume register which is setup as the tone generators.

I have arranged my assembly code to be as simple and as easliy understood as possible. Listing 10-2,needs two inputs,ie the PSG volume 15 to 0 and the PSG register,this is selected from column 4 of table 10-1. These two bytes are added together and bit 7 is set and the data is sent using the technique in cputopsg.

Listing 10-2: Setting the Volume Register.

```
10 SOUND 0,256,0 :REM **    initialise frequency as not
                            defined yet,until 10.5. **

20 POKE    VOL,10 :REM ** volume of 10 **
30 POKE    REG,16 :REM ** select register 1 **

40 CODE

VSTART:    PUSH AF                ; SAVE ANY REGISTERS CORRUPTED
           PUSH BC                ;
           LD A,(VOL)             ; GET VOL
           AND 15                 ; GET RID OF UPPER NIBBLE
           LD B,A                 ; SAVE IT
           LD A,(REG)             ; GET REG
           AND 240                ; GET RID OF LOWER NIBBLE
           ADD A,B                ; VOL + REG
           OR #80                 ; SET BIT 7
           CALL CPUTOPSG          ; SEND TO PSG,see listing 10-1
           POP BC                 ; RESTORE REGISTERS
           POP AF                 ;
           RET                    ; RETURN TO BASIC
VOL:       DS 1                   ; POKE VOLUME HERE
REG:       DS 1                   ; POKE REG HERE

50 STOP
```

You should be able to control the volume of a sound now. The above code and other listings will form a suite of sound utilities which I hope someone can develop into a music composer editor.

10.5 Frequency Synthesis

As already stated the MTX uses a 'pseudo' frequency range,0 to 1024. The actual frequency can be calculated from equation (1).

Actual frequency = N / (32 * f) ....(1)

where:
N = the reference clock frequency,4,000,000 Hz
f = the 'pseudo' frequency

For example,a frequency of 256 Hz gives a pseudo frequency ,f, of:

f = 4,000,000 / (32 * actual frequency of 256) = 488

Therefore in basic the programmer would use a value of 488 to get a frequency of 256 Hz.

The PSG requires 10 bits of information to define the half period of the desired frequency,. This 10 bit frequency,F0 to F9,is loaded into a ten stage tone counter which is decremented at a rate of N/16,where N is the clock speed of the Z80,ie 4,000,000. When the tone counter reaches zero,a borrow signal is produced. This borrow signal toggles the frequency,via flipping over and reloading the tone counter. Therefore the period of the desired freqency is twice the value of the period register.

The PSG has three dedicated tone generator registers 0,2 and 4. The register and frequency are sent to the PSG as two bytes:

```
           MSB                                 LSB
7  6  5  4  3  2  1  0        7  6  5  4  3  2  1  0

1  <-REG->  F3 F2 F1 F0       0  x F9 F8 F7 F6 F5 F4
```

To use this format directly is very confusing,and the extra programming is a pain. The code used to define the frequency and its register has been simplified. I have used the same technique as in basic,ie input the frequency register,ie 0,32,64 or 96(noise register),and then input a pseudo frequency value. Note that two bytes are required to define the frequency,ie

```
           MSB                                 LSB
7  6  5  4  3  2  1  0         7  6  5  4  3  2  1  0

x  x  x  x  x  x F9 F8        F7 F6 F5 F4 F3 F2 F1 F0
```

Listing 10-3: Setting a pseudo frequency of 488 and a volume
of 10.

```
10 GOTO 100

20 CODE

VSTART:                     ; see LISTING 10-2
CPUTOPSG:                   ; see LISTING 10-1

FSTART:     PUSH AF         ;
            PUSH BC         ;
            PUSH HL         ;
            LD HL,(FREQ)    ;
            LD A,L          ; GET F3-F0
            AND 15          ; GET RID OF THE UPPER BITS
            LD B,A          ; SAVE IT
            LD A,(REG)      ; GET THE FREQUENCY REGISTER
            AND 240         ; GET RID OF UNWANTED BITS
            OR #80          ; SET BIT 7
            ADD A,B         ; NOW IN PSG MSB FORMAT
            CALL CPUTOPSG   ; SEND IT
            LD A,L          ; GET F7-F4
            AND 240         ; GET RID OF UNWANTED BITS
            LD L,A          ; SAVE IT
            LD A,H          ; GET F9 AND F8
            AND 15          ; GET RID OF UNWANTED BITS
            LD H,A          ; SAVE IT
            LD B,4          ; SET COUNTER
FLOOP:      SRL H           ; SHIFT H LEFT
            RR L            ; MOVE INTO L AND MOVE LEFT
            DJNZ FLOOP      ; REPEAT UNTIL IN PSG LSB FORMAT
            LD A,L          ;
            CALL CPUTOPSG   ;
            POP HL          ;
            POP BC          ;
            POP AF          ;
            RET             ;
FREQ:       DS 2            ; STORE PSEUDO FREQUENCY

100 POKE REG,16       :REM SELECT VOL REGISTER
110 POKE VOL,10       :REM SET VOLUME
120 RAND USR(VSTART)  :REM CALL VSTART
130 POKE REG,0        :REM SELECT FREQUENCY REGISTER
140 POKE FREQ,232     :REM LSB OF FREQUENCY
150 POKE FREQ+1,1     :REM MSB
160 RAND USR(FSTART)  :REM CALL FSTART
170 STOP
```

The above listing demonstrates how to select frequency and
volume and is equivalent to SOUND 0,488,5 in BASIC.

10.6 Noise Generation

As already mentioned, noise is a random mixture of frequencies which can be used to provide special sound effects like waves, drumbeats,etc. The noise generator consists of a noise source and an attenuator. The noise attenuator is setup as shown in section 10.4. The noise source is actually a shift register with an exclusive OR feedback network. Note that the network has provisions to protect the shift register from locked in the zero state.

Two noise configurations are possible,"periodic" and "white". "Periodic" noise as suggested by the name has a period associated with it, unlike "white" noise which is completely random. To select either noise configuration the Feedback, FB,bit must be either one or zero respectively. The FB bit is bit 2. The PSG requires one byte of information to select the register and the FB and the actual noise selcted. This combined bit is sent to the PSG. The PSG format is:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	1	0	x	FB	NF1	NF0

The upper nibble selects register 6,and doesn't need to specified as this is automatically selected on calling the noise code ,see later. NF1 and NF0 define the shift register and are selected from table 10-2.

Table 10-2: NF patterns and shift rates

NF1	**NF0**	**Shift Rate**
0	0	N/512
0	1	N/1024
1	0	N/2048
1	1	tone 2 output

Therefore the fixed shift rates are derived from the Z80 clock speed. The shift register will only shift at one of the 3 rates as determined by the two NF bits. Note that whenever the noise control register is changed the shift register is cleared.

In one special case though when both NF bits are set,the noise output is directed through tone generator channel 2 . This will allow us to envelope and modulate noise as if it were pure sound. This is necessary to produce drum sounds like the bass drum,etc.

Listing 10-4: Set up white noise with a shift rate of N/512

```
10 GOTO 100

20 CODE

VSTART:                          ; SEE LISTING 10-2
CPUTOPSG:                        ; SEE LISTING 10-1

NSTART:     PUSH AF              ;
            PUSH BC              ;
            LD A,(SHRATE)        ; GET SHIFT RATE
            AND 3                ; GET RID OF UNWANTED BITS
            LD B,A               ; SAVE IT
            LD A,(PORW)          ; PERIODIC OR WHITE NOISE
            AND 4                ; GET RID OF UNWANTED BITS
            ADD A,B              ; FB + NF
            OR 224               ; SELECT REGISTER 6
            CALL CPUTOPSG        ; SEND IT
            POP BC               ;
            POP AF               ;
            RET                  ;
SHRATE:     DS 1
PORW:       DS 1

100 POKE REG,112       :REM SELECT VOLUME REGISTER
110 POKE VOL,10
120 RAND USR(VSTART)
130 POKE SHRATE,0      :REM SHRATE SETUP AS N/512
140 POKE PORW,1        :REM WHITE NOISE
150 RAND USR(NSTART)
160 STOP
```

This code simulates SOUND 3,4,5.

10.7 Sound Off

This last section shows you how to switch off all channels.

Listing 10-5: Switching off all sound channels.

```
10 CODE

VOFF:       LD A,15              ; VOL=OFF
            LD (VOL),A           ; SAVE IT
            LD A,16              ; VOLUME REGISTER 1
VOFF1:      LD (REG),A           ; SAVE IT
            CALL VSTART          ; UPDATE VOLUME
            ADD A,32             ; UPDATE VOLUME REGISTER
            DJNZ VOFF1           ;
            RET                  ; RETURN TO BASIC
VSTART:                          ; SEE LISTING 10-2
CPUTOPSG:                        ; SEE LISTING 10-1
```

MSX Technical Data Appendix

A. Introduction

The VDP Discovered was originally designed as a technical reference manual for the Memotech MTX series. However, since both MSX and Einstein computers share the same IN/OUT mapped device architecture and more importantly use the same CPU (Z80A) and graphics chip (VDP) , the manual now caters for all three computers - MSX , MTX and Einstein.

This appendix is important because it contains the key information to get a number of the listings within the manual to run. The majority of the programs in the manual will work ,however, the MTX and MSX machines use different CPU in/out ports to access the VDP,PSG,etc. This information was extracted from the MSX Technical Reference Document (1984).

However,it is worth pointing out that the MSX standard only pertains to the hardware and to the ROM BIOS call addresses and not to the ROM BIOS code. This effectively means that different MSX manufacturers may use different in/out ports to access the VDP, ie instead of writing VDP data to port #99 ,another manufacturer may use #A5. This point is worth bearing in mind. Therefore you will be required to do a bit of investigation and experimentation. For example, to write to the VDP , use ROM BIOS call #0047. Why not disassemble the code at this address. It is worth obtaining a copy of the MSX ROM BIOS calls,usually the MSX user groups are the best source.

As you will soon see, I have only included the essential information needed to get the ball rolling. The text in the manual is self explanatory and it should only be a matter,in chapters 2-8, of changing the MTX port addresses and one or two MTX BIOS calls to their MSX equivalents. Chapters 1,9,and 10 are less straight forward. It is hoped that you read chapters 2 to 8 to get a grounding in writing practical Z80 code and that by the end of these chapters you will be writing new routines like drawing lines. If you have managed this, then you will probabely know what to do with the RAM/ROM mappings,keyboard and PSG data included in this appendix.

B. MSX Hardware Specification

B.1 Devices

CPU	-	Z80A running at 3.5795 MHz , 1 wait state in M1 cycle.
VDP	-	TI TMS 9918A Compatible.
PSG	-	GI AY-3-8910 Compatible.

B.2 Printer Port

See chapter 8 for a more detailed discussion of printers and screen dumps. Sending data to a printer can be acheived in one or two ways:

(a) using the assigned in/out ports which will involve writing your own code or
(b) using the MSX ROM Bios code accessed via the Bios call address.

I will describe both ways briefly. The MSX machines use ports #90 and #91 to check the printer's status and to send data to the printer, table b-1, briefly summarises the function of these ports.

Table b-1: Ports used in printing

Register	**Read/Write**	**Description of Port Function**
#90	Read	Bit 1 indicates the busy status.
#90	Write	Bit 0 for strobe output.
#91	Write	Print data.

When we say the port or register is used to READ data then use the Z80 assembly language command IN A,(port or register) and for writing use OUT (port or register),A . The code below has not been checked but the code for sending either text or graphic data to the printer will be something similar:

Listing b-1: Sending a text message to the printer (hypothetical)

```
prt:       IN A,(#90)            ; read port #90
           BIT 0,A               ; test if status busy
           JR Z,PRT              ; if busy try again, else
           LD A,(prtchar)        ; get character to print
           OUT (#91),A           ; send data
           LD A,1                ; strobe output to printer
           OUT (#90),A           ; send to port #90
           RET                   ;

prtchar:   DB 65                 ; print the character A
```

Listing b-2: Using the MSX ROM bios calls to do a similar job:

```
romprt:    LD HL,prtdata         ; where the message to be printed
                                 ; is located.
           LD B,15               ; length of message
loop:      CALL #00A8            ; checks the printer status.
           JR Z,loop             ; Z flag set if not ready
           LD A,(HL)             ; Reg A holds the MSX/ASCII
                                 ; character code when sending text or
                                 ; graphic data if in graphics mode.
           CALL #00A5            ; this sends the data to the printer.
           DJNZ loop             ; repeat until message printed and
                                 ; exit.
           RET                   ; end of printer routine.

Prtdata:   DB "printed message"
```

Why not disassemble the code at #00A5 and #00A8.

B.3 Screen Display

The MSX at switch on is configured to give the following resolutions according to the screen mode,see table b-2.

Table b-2: Screen Mode resolutions at switch on

Mode	Dot resolution	Character resolution	Sprites
GI	240 x 192	30 x 24	Yes
GII	240 x 192	30 x 24	Yes
TXT	240 x 192	40 x 24	No
MC	64 x 40	30 x 24	Yes

The 8 pixels from the left and right of the horizontal are not used by the ROM Bios software. However, by programming the VDP from Z80 assembly language, you can use the full screen width. The MSX uses ports #98 and #99 to read and write to the VDP. This is fully explained in VDP Discovered. Simply replace all the references of #01 or #02 when used with OUT or IN commands with #98 and #99 respectively . At the end of chapter 2 there are one or two MSX screen mode mappings. However,the VRAM mappings according to the MSX technical reference document are slightly different and are given below for completeness.

Table b-3: Screen mode VRAM mappings

Description	TEXT	GI	GII	Multicolour
Name Table	#0000	#1800	#1800	#0800
Colour Table	N/A	#2000	#2000	N/A
PGT	#0800	#0000	#0000	#0000
SAT	N/A	#1B00	#1B00	#1B00
SPGT	N/A	#3800	#3800	#3800

B.4 Keyboard

The majority of the MTX listings in the VDP Discovered use the MTX Rom call #0079 to read the keyboard.

Listing b-3: MTX ROM Bios keyscan code

```
mtxkey:     XOR A       ; clear register A. Call #0079 scans the kbd
            CALL #0079 ; matrix then exits,setting the Z-flag if no
            JR Z,mtxkey; key pressed or storing in A the ASCII code
            RET         ; of the key pressed.
```

MSX users have the ability to do likewise with the MSX ROM call #009F. Notice from listing b-4,that the MSX keyscan code only returns when a key is pressed therefore no need to loop as in MTX case.

Listing b-4: MSX ROM Bios keyscan code

```
msxkey:     XOR A       ;
            CALL #009F ; waits for a character to be typed at the kbd
            RET         ; before exiting the call.
```

B.5 Sound

MSX uses a completely different sound chip, the AY-3-8910, than the TI SN 76489A of the MTX . The layout and the registers of this AY chip are given in tables b-4 & b-5. The CPU accesses the sound registers via the in/out ports shown in table b-6.Note that there are a number of MSX ROM Bios calls to access the PSG, so consult your ROM Bios calls list and disassemble the code to get an idea as to how to program it.

Table b-4: PSG Registers & Functions

Register		**Bit information**							
Number	**Description**	**b7**	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**
R0	Tone A	<------ 8-bit Fine Tune on Channel A ------>							
R1	Tone A	0	0	0	0	<- Coarse Tune on A ->			
R2	Tone B	<------ 8-bit Fine Tune on Channel B ------>							
R3	Tone B	0	0	0	0	<- Coarse Tune on B ->			
R4	Tone C	<------ 8-bit Fine Tune on Channel C ------>							
R5	Tone C	0	0	0	0	<- Coarse Tune on C ->			
R6	Noise	0	0	0	<- 5-bit period Control ->				
R7	Enable	in/out		noise			tone		
"	"	i/ob	i/oa	NC	NB	NA	TC	TB	TA
R10	Amplitude A	0	0	0	M	L3	L2	L1	L0
R11	Amplitude B	0	0	0	M	L3	L2	L1	L0
R12	Amplitude C	0	0	0	M	L3	L2	L1	L0
R13	Envelope	<--------- 8-bit Fine Tune E ----------->							
R14	Period	<--------- 8-bit Coarse Tune E ----------->							
R15	ENV shape	0	0	0	0	cont	att	alt	hold
R16	I/OA Data	<------ 8-bit parallel I/O on Port A ------>							
R17	I/OB Data	<------ 8-bit parallel I/O on Port B ------>							

Register 7 is a very important register. It is the Mixer register and as such decides what we hear. Each bit can either be set on (1) or off (0). The combination of these bits determines the sound output. Note that bits 6 and 7 don't affect the sound as these are in/out device selectors. Bits 3,4,5 are used for setting channels A,B and C to that of a noise as needed for drum and other special sound effects. Bits 0,1 and 2 are used for setting channels A,B and C to sounds defined by the fine and coarse tones and the amplitude registers. The only restriction is that you cannot have both noise and tone selected for the same channel,therefore if you want channel A to be noise then bit 3 is set to 1 and bit 0 is reset to 0.

Register pairs R1/R0 ,R3/R2 and R5/R4 determine the pitch or tone of the sound. The register pairs combine to give a 16-bit register like that of H & L registers of the Z80. However,the top 4 bits of the most significant register (R1,R3 or R5) are ignored, thus giving a tone range of 0 to 2^12 (0 to 4095),see figure b-1. The three amplitude channels determine how loud the sound is. Bits 0-3 or L3 to L0 are used for volume control. The volume ranges from 0 to 15. The M bit of R10, R11 or R12 when reset (0), means that the tone the amplitude register refers to, gives a sound which is normally associated with most computers, ie the beep.

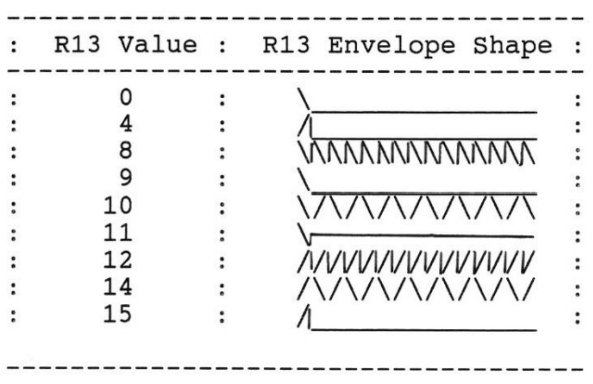
R1/R3/R5 (msb)								**R0/R2/R3** (lsb)							
0	0	0	0	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

Figure b-1: The pitch (tone) of the sound, range 0 - 4095.

However,when the M bit of either R10,R11 or R12 is set (1),then the associated channel either A,B or C,will adopt the envelope shape, set by R15. Note that cont, att, alt, and hold of Register 15 define the envelope of the sound. The predefined envelopes or waveforms of the sound are like that of a piano key being struck or can be a completely different sound,for instance, envelope shape 10, which is very much like that of a police siren where the sound rises and falls and then repeats over and over again,see table b-5.

The register pair R13/R12 is the envelope sustain on all channels. This is a full 16-bit register pair offering a range of 0 to 65535. This means that the envelope waveform defined by R15 when selected will be repeated x number of times,where x is the number held in R13/R12.

Table b-5: the PSG Envelopes



R13 Value	R13 Envelope Shape
0	_______________
4	/|_______________
8	\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
9	_______________
10	\\/\\/\\/\\/\\/\\/\\/\\
11	\\‾‾‾‾‾‾‾‾‾‾‾‾‾‾
12	/|/|/|/|/|/|/|/|/|/|/|/|/|
14	/\\/\\/\\/\\/\\/\\/\\/
15	/|_______________

The CPU access the PSG through ports #A0,#A1 and #A2,see table b-6, see chapter 10 of VDP Discovered,for an analogous system.
The MSX PSG offers the programmer more power and greater features than the MTX PSG.

Table b-6: CPU registers used to access the PSG registers

CPU Register	**Read/Write**	**Description**
#A0	Write	Address latch
#A1	Write	Write data to PSG
#A2	Read	Read data from PSG

5

B.6 Memory Map

The MSX BASIC rom is located at ram #0000 to #7FFF. #FFFF to #C000 contains the 16k of RAM or on 64k systems, #FFFF to #8000 contains 32k of user RAM. As you can see from figure b-2, the basic unit has 4 logical areas or slots (0-3). Each slot is 64k wide. Therefore the total memory space can be expanded to 256k. These 4 logical slots can be expanded futher to give 4 physical slots per logical slot. Now the memory space can be further exapnded to 1 mb.

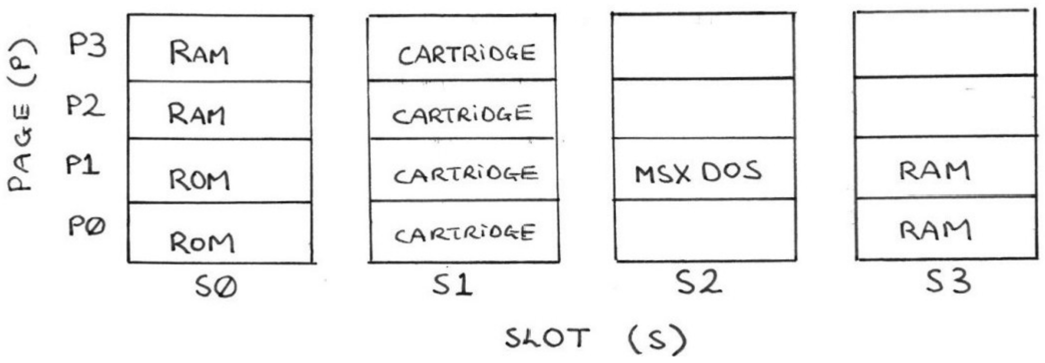


Figure b-2: MSX Memory Map.

Z80 CPU port #A8 is used to access the slot select register. To map the physical memory space to the logical CPU memory spcae is carried out in 16k chunks or pages. It is at this point that the MTX definitions & MSX ones differ:

MTX page = MSX slot   MTX block = MSX page

See figure b-3 for an example.

```
  b7  b6  b5  b4  b3  b2  b1  b0
 --------------------------------
: 0 : 0 : 1 : 0 : 0 : 0 : 0 : 0 :
 --------------------------------
  slot 0  slot 2  slot 0   slot 0
    on      on      on       on
  page 3  page 2  page 1   page 0
```

each 2 bit combination gives a possible 4 choices or slots:

```
0 0 = 0
0 1 = 1
1 0 = 2
1 1 = 3
```

Figure b-3: Slot selections example.

C. MSX Data Load,Save and Run

MSX users will require a Z80 assembler. Hisoft offer an excellent Z80 assembler called DEVPAC. The advantages offered by this assembler over many of their rivals is that it is available for Einstein, Memotech, ,MSX ,Amstrad and CPM. All code and libraries developed can be moved to any of the other computers without having compatibility problems also the interface and commands remain the same.

In many of the examples within VDP Discovered, the command VS number has been used. This is the MTX BASIC command for Virtual Screen or the Screen mode. VS 4 is VDP graphics mode II which is equivalent to MSX BASICs command SCREEN 2. VS 5 is VDP TEXT mode or SCREEN 0 in MSX BASIC.

At random, listing 5-4 (pages 47-48) was selected to demonstrate how easy it is to run this listing on a MSX machine. First of all load in the Z80 assembler text editor and type in the following source text:

```
ORG #9000   ( or &9000 or 9000H )

; main program


COLRES:     LD HL,VDPREGGII
            CALL VDPREGSET8
            LD HL,8192
            CALL VDPWRTSEL
            XOR A
            EX AF,AF'
            XOR A
            LD BC,6144
COLLOP:     EX AF,AF'
            OUT (#98),A
            INC A
            CP 16
            JR NZ,NORESET
RESETCOL:   XOR A
NORESET:    DEC BC
            EX AF,AF'
            LD A,B
            OR C
            JR NZ,COLLOP
            RET


; program subroutines


VDPWRTSEL:  PUSH AF
            LD A,L
            OUT (#99),A
            LD A,H
            OR #40
            OUT (#99),A
            POP AF
            RET
```

```
VDPREGSET8:LD BC,#0800
REGWRTVDP: LD A,(HL)
           OUT (#99),A
           LD A,C
           OR #C0
           OUT (#99),A
           INC C
           INC HL
           DJNZ REGWRTVDP
           RET


; program variables


VDPREGGII: DB #02,#C2,#06,#FF,#03,#36,#07,#F5
```

Now save to DISC or TAPE accordingly using the TEXT editor command to save. Compile this Z80 source text into Z80 machine code using the Z80 assembler. This should result in either a COM or BIN or OBJ file.
Reset the MSX to BASIC and type in the following MSX BASIC listing:

```
10 CLEAR &H9000
20 BLOAD "fname",&H9000
30 SCREEN 2: CLS
40 DEF USR0=&H9000
50 A=USR0
60 GOTO 60
```

Type in RUN <ent> to execute the program, you should see a very colourful screen,highlighting the VDP's colour resolution in GII mode.
To exit back to MSX BASIC ,press the break key.

The MSX like the Einstein and Memotech default to TEXT mode (Screen 0) after exery BASIC program is RUN. This means, as in the example above , if you do not include Line 60, you will miss the colourful effect of the machine code as the system defaults back to TEXT mode the instant the machine code is executed. The effect of the Z80 machine code will only be on the screen for a second , if that , as the system switches from Graphics mode II (the desired mode for this listing) back to TEXT mode.

Tatung Einstein Technical Data Appendix

D. Introduction

This short appendix, summarises the key technical data needed to take advantage of the many listings within "VDP Discovered". The Einstein port numbers for the screen, keyboard and sound access are supplied with the relevant text and where required, listings are used to give extra clarity. I would like to thank the UKEUG for providing the technical information .

E. Einstein Hardware

E.1 Devices

CPU - Z80A
VDP - TI TMS 9918A Compatible
PSG - GI AY-3-8910 Compatible

E.2 Screen Display

The Einstein screen is configured like that of the Memotech. This means that both and the text screen and the graphics mode II screens can be configured so that no overlap of screen data in VRAM occurs , in other words the integrity of both screens are retained. Swapping between the two screens requires the desired screen mode to be selected (VDP registers 0 and 1 , see table 2-4 , page 12) and that the name table is pointing to either the Graphics II Name Table or the TEXT Name Table (where Name Table is interchangeable with the word for SCREEN), see Example below:

Table e-1: Einstein VDP register Values for GII and TEXT screen modes.

Mode/VDP Reg Num	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**
Einstein GII	02	C2	0E	FF	03	76	03	F4
Einstein TEXT	00	D2	0F	FF	03	76	03	F4

As already stated in Chapter 2, the Einstein uses ports 8 and 9 to communicate between the CPU and the VDP/VRAM.

The booklet "Einstein Compendium" from the UKEUG has a summary chapter on the Einstein VDP configuration,with a number of examples on Screen Loading and Saving, Fonts and a character designer and/or refer to chapters 2 - 7 within, for a more detailed discussion of the VDP.

E.3 Keyboard

The majority of the MTX listings use the MTX ROM call #0079 to read the keyboard, see listing e-1.

Listing e-1: MTX ROM Bios Key scan code

```
MTXKEY:     XOR A
            CALL #0079
            JR Z,MTXKEY           ; loop until key pressed
            RET
```

Einstein users have the ability to do likewise with the Einstein Machine Call (RST). Notice the difference in terminology. Let me explain. The Z80A CPU has a number of special one-byte predefined CALL addresses. These are called ReStarTs. Normally the Z80 mnemonics for calling a subroutine would be:

```
CALL #0008 ; when assembled this would stored as 3 - bytes
           ; CD 08 00 . The Z80A requires 5 cycles or
           ; 8.5 usecs @ 2MHz  to execute this command.
```

On the other hand a Restart which has the mnemonic RST, is a lot quicker than this at only 5.5 usecs @ 2MHz. The above call would be coded as:

```
RST #08    ; one byte would be #CF.
```

The Z80A has 8 of these Restarts:

```
RST #00 ( CALL #0000 ) ; #C7 RST #08 ( CALL #0008 ) ; #CF
RST #10 ( CALL #0010 ) ; #D7 RST #18 ( CALL #0018 ) ; #DF
RST #20 ( CALL #0020 ) ; #E7 RST #28 ( CALL #0028 ) ; #EF
RST #30 ( CALL #0030 ) ; #F7 RST #38 ( CALL #0038)  ; #FF
```

Usually OS designers will use these Restarts for quickly accessing Graphics or error messages or etc.

On the Einstein RST #08 is used a lot for the main MOS (Machine Operating System) functions like handling the screen or the keyboard or the printer. The RST #08 command is usually followed by a DataByte ,ie DB #9F. The DataByte tells the MOS which BIOS call you would like to access, for the above example, we are selecting to send data to the printer. I refer you to The "Tatung Einstein User" magazine ,vol 1 ,num 4 for a full list of MCALs or RSTs for the Einstein MOS.
To scan the keyboard the Einstein user has two options:

1. To scan the keyboard, setting Z-flag to zero for a valid key and then to continue with the rest of the program.
2. To scan the keyboard until a key is pressed. Again the Z flag is set to zero if keypressed.

In both cases, Register A holds the result of the keypress,in ASCII format , ie 67 for the Letter 'C'.

Option 1 (ZRSCAN) works in a similar manner to the MTX BIOS as shown in Listing e-2. Option 2 (ZKEYIN) is more like the MSX Bios where the code waits until a key is pressed, see listing e-3.

Listing e-2: Using ZRSCAN to read Keypress on Einstein

```
EINKEY1:   XOR A
           RST #08
           DB #9B
           JR Z,EINKEY1
           RET
```

Listing e-3: Using ZKEYIN to read Keypress on Einstein

```
EINKEY2:   XOR A
           RST #08
           DB #9C
           RET
```

E.4 Sound - Programmable Sound Generator (PSG)

The MSX technical data appendix, section B.5 , gives all the relevant text about the AY-3-8910 PSG. The only change being that of table b-6 , table e-2 gives the port numbers the CPU on the Einstein uses to access the PSG.

Table e-2: CPU registers used to access the PSG registers

CPU Register	**Read/Write**	**Description**
#02	Write	Address latch
#03	Write	Write data to PSG
#02	Read	Read data from PSG

E.5 Printer Port

As you will have noticed if you have read the MSX technical data appendix, this section appears to be out of sequence. The reason for this is that a number of other points like RST and PSG needed to be covered before this section could make sense.

Chapter 8 within and "Einstein Compendium" and the "Einstein User magazine ,vol 1,num 4 " are useful references for a more detailed discussion of printers and the software needed for screen dumping.

The PSG, mentioned earlier , has two registers not associated with the generation and control of the sound. These two registers are mapped registers, in the same vain as the Z80 CPU In/Out mapped ports. Register 16 is used as an 8-bit Parallel In/Out (PIO) port, which the Einstein designers used for the printer port. Register 7 of the PSG requires that bit 6 be set before the register 16 can be used.

The Z80 CPU has to communicate with the PSG to tell it to enable register 16 (setting bit 6 of register 7) and to read and write to register 16, see table e-3.

Table e-3: CPU - PSG communication ports for the printer.

CPU Port Num	PSG Reg Num	R/W	Description
#31	R7	Write	Printer Control Reg
#30	R16	Read	Printer Data Reg
#30	R16	Write	Printer Data Reg
#20		Read	Printer Status

Only bits b2,b3 and b4 of port #20 are used for the printer status, all other bits can be ignored:

b2 = "BUSY"   b3 = "PAPER EMPTY"   b4 = "ERROR"

The MSX computers use this and the other PIO (parallel In/Out port) similarly to the Einstein.

Sending data to a dot matrix printer can be programmed in one or two ways :

(a) using the assigned In/Out ports, which will involve you in writing more complex code, see Listing e-4.

(b) using the Einstein BIOS code accessed via the MOS calls. This requires less programming effort as a number of the key subroutines are already available,see Listing e-5.

Listing e-4: User Printer Dump Example (hypothetical)

```
USERPRT:   LD HL,PRTDATA          ; where the message is stored
UPLOOP:    CALL CHKSTAT           ; test printer status
           LD A,(HL)              ; get message, byte at a time
           CP #FF                 ; if #FF then message printed
           RET Z                  ; and exit
           PUSH AF                ; save message byte
           XOR A                  ; clear reg A
           SET 6,A                ; set bit 6 of PSG
           OUT (#31),A            ; PIOA enabled
           POP AF                 ; restore message byte
           OUT (#30),A            ; byte ready for sending
           IN A,(#30)             ; strobe data to printer
           JR UPLOOP              ; keep looping until finished.
PRTDATA:   DB "Test Printer",#FF

CHKSTAT:   IN A,(#20)             ; get printers status
           AND #1C                ; ignore all bits except 2,3 and 4
           CP #10                 ; is printer okay for transmission
           JR NZ,CHKSTAT          ; loop if not
           RET                    ; else okay to proceed.
```

Listing e-5: Einstein MOS version

```
MOSPRT:    LD HL,PRTDATA          ;
MPLOOP:    CALL CHKSTAT           ;
```

```
           LD A,(HL)              ;
           CP #FF                 ;
           RET Z                  ;
           RST #08                ; MOS call
           DB #9F                 ; printer send routine
           JR MPLOOP              ;

PRTDATA:                          ; as for listing e-4

CHKSTAT:                          ; as for listing e-4
```

E.6 Memory Map

Unlike the MSX and the Memotech which require 32k and 16k respectively for the ROM BIOS code and the BASIC Interpreter, the Einstein is configured like a CPM computer, with only a small amount of ROM bootstrap code using RAM, the remaining 64k is available to the programmer. The Einstein user loads all the necessary DOS and MOS information from Disc.

The advantages of this approach is that new versions are easily implemented and cheaper whereas MSX and Memotech users would require a more costly ROM replacement. ROM designers are limited in coding by the size of the ROM. Note for Taped based computers ,like the MSX, it is essential to have the OS on ROM ,as it would take an enternity to load from tape.

All Applications from BBC BASIC, Logo to wordprocessing are all loaded into RAM if and when required. The documentation supplied with these programs should give memory configuration of the Einstein after the application has been loaded into RAM.

F. Einstein Data Load,Save and Run

Einstein users will require a Z80 assembler. Hisoft offer an excellent Z80 assembler called DEVPAC. The advantages offered by this assembler over many of their rivals is that it is available for Einstein, Memotech, ,MSX ,Amstrad and CPM. All code and libraries developed can be moved to any of the other computers without having compatibility problems also the interface and commands remain the same.

In many of the examples within VDP Discovered, the command VS number has been used. This is the MTX BASIC command for Virtual Screen or the Screen mode. VS 4 is VDP Graphics mode II and VS 5 is VDP TEXT mode on the Memotech.

At random, listing 5-4 (pages 47-48) was selected to demonstrate how easy it is to run this listing on a Einstein machine. First of all load in the Z80 assembler text editor and type in the following source text:

```
ORG #9000   ( or &9000 or 9000H )

; main program


COLRES:     LD HL,VDPREGGII
            CALL VDPREGSET8
            LD HL,8192
            CALL VDPWRTSEL
            XOR A
            EX AF,AF'
            XOR A
            LD BC,6144
COLLOP:     EX AF,AF'
            OUT (#08),A
            INC A
            CP 16
            JR NZ,NORESET
RESETCOL:   XOR A
NORESET:    DEC BC
            EX AF,AF'
            LD A,B
            OR C
            JR NZ,COLLOP
            RET

; program subroutines


VDPWRTSEL:  PUSH AF
            LD A,L
            OUT (#09),A
            LD A,H
            OR #40
            OUT (#09),A
            POP AF
            RET
VDPREGSET8:LD BC,#0800
REGWRTVDP:  LD A,(HL)
            OUT (#09),A
            LD A,C
            OR #C0
            OUT (#09),A
            INC C
            INC HL
            DJNZ REGWRTVDP
            RET

; program variables

VDPREGGII:  DB #02,#C2,#0E,#FF,#03,#76,#03,#F4
```

Now save to DISC accordingly using the TEXT editor command to save. Compile this Z80 source text into Z80 machine code using the Z80 assembler. This should result in either a COM or BIN or OBJ file. Reset the Einstein to BASIC and type in the following Einstein BASIC listing:

```
10 CLEAR &9000
20 LOAD "fname.com" : REM  or "fname.bin"  or "fname.obj"
30 CALL &9000
40 GOTO 40
```

Type in RUN <ent> to execute the program, you should see a very colourful screen,highlighting the VDP's colour resolution in GII mode. To exit back to Einstein BASIC ,press the break key.

The Einstein like the MSX and Memotech default to TEXT mode after every BASIC program is RUN. This means, as in the above example , if you do not include line 40 , you will miss the colourful effect of the machine code as the system defaults back to TEXT mode the instant the machine code is executed. The effect of the Z80 machine code will only be on the screen for a second ,if that , as the system switches from Graphics mode II (the desired mode for this listing) back to TEXT mode.

Miscellaneous Appendix

G. **Figures Reference**:

H. **Book List**

H.1 Memotech MTX,SDX and FDX

New Memotech Manual	by Pheonix Publishing
Memotech Computing	by Ian Sinclair
MTX Tape to Disc Conversion Booklet	by AFW Software
Introduction to Assembly Language	by Graysoft
Introduction to BASIC Programming	by Graysoft
Midi Projects	by R.A.Penfold
Computer Music Projects	by R.A.Penfold

H.2 MSX

MSX	by Teach Yourself
Behind the Screens of the MSX	by M.Shaw
The MSX	by T.Marriot
Starting Machine Code on the MSX	by G.P.Rodley
Ideas for the MSX	by K.Zetie
Programming in MSX Basic	by Avalon Software
The MSX Red Book	by Avalon Software
Microsoft Technical Document	by Microsoft
Midi Projects	by R.A.Penfold

H.3 Tatung Einstein

Albert Revealed	by Crystal Research
Einstein Compendium	by UKEUG
Einstein Hardware Manual	by Tatung
BBC (Z80) Basic Manual	by Tatung
Using Dr Logo on the Einstein	by Tatung

H.4 MTX,MSX and Einstein

Power Graphics (Graphics Toolbox)	by AFW Software
VDP Discovered ($)	by AFW Software
Programming the Z80	by R.Zaks
VDP Programmers Guide	by Texas Instruments

($) = previously released as Advanced Reference Manual for the Memotech MTX Series. Renamed as VDP Discovered.