THE MAGIC OF MSX

BY BERNARD L. BURKE

INTRODUCTION

This book is not a games book — there are plenty of those on the market already — THE MAGIC OF MSX is a book for the person who knows some basic programming and is ready to advance both in basic and machine code. The book is divided into two main parts:

PART 1 consists of chapter 1 through to chapter 15. This part deals with the memory map, the video chip, the system variables and other useful information for the basic programmer. A knowledge of the information contained in part 1 is essential for the machine code programmer.

PART 2 starts at chapter 16 and deals with machine code programming on the MSX. You will find within these pages the tools you need to enter the magic world of machine code. These tools include a full machine code assembler and details of many ROM routines to assist you in your programs. There are also many source files to illustrate the SUPER ASSEMBLER operation and the operation of the ROM routines.

LISTINGS

How many times have you bought a book full of listings and then found most of the listings full of errors?

Disheartening isn't it?

We have tried to avoid that problem by providing a tape containing all the listings. You should have received the tape when you bought the book — if you did not get the tape then consult your dealer.

BRICKBATS AND BOUQUETS

This book is written by an MSX USER for other MSX USERS. We want the book to be accurate and to provide the information which is required by the reader.

We would appreciate your comments, suggestions, or criticisms about this book so that future editions can reflect your needs.

Send your comments to the publishers:

INTERSOFT (PTY) LTD.,
P.O. BOX 5078,
JOHANNESBURG 2000,

THANKS

Thanks are due to INTERSOFT for publishing and distributing this book. Thanks also to all the people who phoned me on the MSX HOTLINE — many of the ideas in the book were sparked off by the questions you asked.

A special thank you to BENNIE VAN DER MERWE who wrote the SUPER ASSEMBLER — well done BENNIE.

Finally, a special thank you to my wife DOROTHY and the children (MATTHEW, MARK, SARAH, and LUKE) for their encouragement during the long months of writing.

USING THIS BOOK

We suggest that you read through the book fairly quickly to gain an appreciation of the contents and then start to work through the chapters thoroughly from the beginning.

Use the tape supplied — you will find the listings on the tape in the same order as they appear in the book. NOTE that for all listings you type CLOAD followed by ENTER and then PRESS PLAY ON THE TAPE.

There will be a great temptation to leap immediately into machine code but please cover the earlier sections first — you need to know about, for example, the memory and the video chip before doing any serious machine code work.

Finally when you have worked through the book keep it near the computer for reference purposes — the appendices will be of particular use in this regard.

HAPPY COMPUTING

Cover Design: Susan Woolf

Typesetting and Printing: Minit Print, Medical Centre.

ISBN © 620 09070 7

INTERSOFT
P.O. Box 5078,
Johannesburg, 2000,

Tel: (011) 337-5806/7
Telex: 48-3868 SA.

TABLE OF CONTENTS

CHAPTER 1

NUMBER SYSTEMS

This chapter has been included to introduce the reader to the NUMBER SYSTEMS used by the computer. If you are already familiar with the concepts presented here then skip this chapter and continue with chapter 2 — you may however want to glance through chapter 1 to refresh your knowledge of number systems.

The computer reduces all data (even text, music, and graphics) to a series of numbers which can be stored in memory and easily manipulated by the micro processors which make up the computer.

We are all familiar with the decimal system which is used all the time by everyone — the computer however finds the decimal system very difficult to work with. This is because of the nature of the memory and processor chips within the computer.

Your MSX Machine, in common with other computers, mostly uses the BINARY number system which counts up in 2's (DECIMAL counts in 10's). The MSX also uses HEXADECIMAL (counts in 16's) and OCTAL (counts in 8's) — the computer makes limited use of the familiar DECIMAL system.

NOTE that the "COUNT" of a number system is known as the number BASE — so, for example, HEXADECIMAL or HEX has a number BASE of 16 and BINARY has a BASE of 2.

Lets look at NUMBER SYSTEMS:

THE DECIMAL SYSTEM

Consider the following example taken from the packing shed of a peach distributor. This company used the following packaging system:

a) Peaches were packed into trays — 10 peaches to the tray.

b) Trays were packed into boxes — 10 trays to the box.

c) Boxes were packed into cartons — 10 boxes to the carton.

d) Cartons were crated — 10 cartons to the crate.

At the end of the day the packing foreman had to report the number of peaches packed that day — he calculated this by counting the number of trays packed and multiplying by 10.

One day the foreman made a wondrous discovery — that day 76540 peaches had been packed and he noticed that each digit of the number had a significance which he had never before recognised:

7 full crates had been packed.
6 uncrated cartons were full.
5 boxes were full.
4 trays were full.
0 peaches left over.

Our hero had discovered the basic principles of the decimal system — that each digit in a decimal number represents a number of "LOTS" and the size of each "LOT" is indicated by the position of the digit within the decimal number.

Lets examine this in more detail — we were taught at school that a number is made up as follows:

TABLE 1.1

ten thousands	thousands	hundreds	tens	units
7	6	5	4	0

Examine table 1.1 closely and you will find that the value of a "LOT" is ten times the value of the "LOT" immediately to the right of the "LOT" under consideration.

Decimal is a number system with a BASE TEN and so we can say that in the case of decimal a particular "LOT" value is equal to the value of the "LOT" on the right times the number BASE.

Now consider the following which is another way of depicting the decimal system:

TABLE 1.2

10000's	1000's	100's	10's	1's
10^4	10^3	10^2	10^1	10^0

The value of any "LOT" is equal to the NUMBER BASE raised to the power of the NUMBER POSITION. The number position is counted from right to left with the right hand digit being in position zero.

NOTE:

a) Any number raised to the power of zero is equal to one and so the value of the "LOT" in the right hand number position is always equal to one.

b) The digit in any number position can range from 0 to the number base minus 1.

c) The value of any particular number position is equal to the digit in that position multiplied by the "LOT" value at that position.

BINARY NUMBER SYSTEM

The BINARY NUMBER SYSTEM has a BASE of 2 — this means that a number position will always contain the digit 0 or 1 (digits range from 0 to the number base minus 1). The binary system is depicted in the following table:

TABLE 1.3

128's	64's	32's	16's	8's	4's	2's	1's
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

TABLE 1.3 describes the 8 smallest "LOTS" of a binary number — remember that with the binary system a digit can range from zero to one and so any particular "LOT" is either present or absent. Such a number (8 "LOTS" OR BITS) can range from zero to 255. Notice how all the principles which applied to decimal numbers also apply to binary numbers — only the number base has changed.

The binary system is particularly suited to the computer because the computer must only remember whether a BIT is on (1) or off (0) thus indicating whether a "LOT" is present or absent.

EXERCISE 1

Lets convert the decimal number 156 into binary:

To do this we extract binary lots and set the binary bits as required — starting with the senior (most significant) bit and moving through to the least significant (junior) bit. Work through the following table to understand the conversion method.

TABLE 1.4

decimal remainder	binary lot	binary digit
156	128	1
156 — (128*1) = 28	64	0
28 — (64*0) = 28	32	0
28 — (32*0) = 28	16	1
28 — (16*1) = 12	8	1
12 — (8*1) = 4	4	1
4 — (4*1) = 0	2	0
0 — (2*0) = 0	1	0

So decimal 156 = binary 10011100

Now calculate the binary equivalent of 241 and 65 using the tabulation method.

BIN$ FUNCTION

MSX basic provides a simple way to convert from decimal to binary using the BIN$ function.

try — PRINT BIN$ (241) — press ENTER

The computer responds with 11110001 — did you get that by the tabulation method?

now try — PRINT BIN$ (65) — press ENTER

The computer responds with 1000001 — only 7 digits! must be something wrong!

The reason for this is that the computer does not print leading zeros in a binary number. To get over this problem use the following code to convert to 8 bit binary:

```
A$ — BIN$ (65): A$ = STRING$ (8—LEN(A$),48) + A$:
PRINTA$
```

This time the computer prints 01000001 — thats better!

The computer does all its internal calculations and storage in binary format and it is therefore often convenient for the programmer to work in binary as well. We have seen how binary numbers consist of long strings of 1's and 0's which is fine for the computer but difficult for humans — because of this the HEXADECIMAL number system was developed.

HEXADECIMAL NUMBER SYSTEM

HEXADECIMAL or HEX is a number system with a base of 16 which is compatible with the binary system but can represent larger numbers using less digits. One HEX digit is equivalent to 4 BINARY digits.

In common with other numbers a HEX digit must range from zero to the number base minus 1. This means that the hex digit must range from zero to 15 — seems like a problem for a single digit. This problem is overcome by using letters A — F to represent digits of value 10 to 15.

TABLE 1.5

4096's	256's	16's	1's
16^3	16^2	16^1	16^0

Table 1.5 describes the "LOTS" of a hex number which can range from 0 to 65535 decimal or from 0 to FFFF hex.

Table 1.6 shows a comparison of decimal, binary and hex – please study the table carefully so that you fully understand the relationship between the different number systems. Note in particular that a single HEX digit represents a 4 BIT binary number. Incidentally a single HEX digit is often known as a NIBBLE.

TABLE 1.6

DECIMAL/HEX/BINARY TABLE

DECIMAL	HEX	BINARY
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Using TABLE 1.6 you can convert any binary number into HEX. Proceed as follows:

a) Using leading zeros ensure that the number of digits in the binary number is exactly divisible by 4.

b) Separate the binary digits into groups of 4.

c) Using table 1.6 convert each group of 4 binary digits into a single hex digit.

This method is illustrated in the following example:

EXAMPLE

decimal 241 = binary 11110001

binary 11110001 = 1111 0001 ...... separate into groups of 4.

binary 11110001 = F 1 ...... hex conversion from table 1.6.

binary 11110001 = hex F1 ...... solution.

Now you try to convert decimal 138 to binary and then to hex. You should get the result —

decimal 138 = binary 10001010 = hex 8A.

HEX$ FUNCTION

MSX basic provides the HEX$ function for conversion of numbers into HEX.

try — PRINT HEX$ (201) — result C9.

The computer does not print leading zeros and so if you require a hex number with say 4 digits you should use the following code:

```
A$ = HEX$(201):A$ = STRING$(4—LEN(A$),48)+A$:PRINTA$
```

Now the result is 00C9 — a hex number of 4 digits as required.

BINARY/HEX TO DECIMAL CONVERSION

binary to decimal — PRINT &B00110111 — result 55

hex to decimal — PRINT &H37 — result 55

THE OCTAL NUMBER SYSTEM

The last number system used by the computer is the OCTAL system which has a base of 8.

TABLE 1.7 describes the OCTAL system.

TABLE 1.7

512's	64's	8's	1's
8^3	8^2	8^1	8^0

The basic function OCT$ is used to convert decimal numbers into octal and the prefix &o is used to convert octal into decimal.

PRINT OCT$(156) ...... result 234 octal

PRINT &o361 ...... result 241 decimal

****

That's all about number systems — in the next chapter we will examine some of the well known but little explained computer terms.

CHAPTER 2

BITS BYTES AND OTHER WONDERFUL THINGS

In CHAPTER 2 we examine a few computer terms and conduct an interesting little exercise using PEEK and POKE, but first lets look at how the MSX manufacturers define machine size. The size of a computer is a measure of its user memory capacity — the MSX standard calls for a minimum of 8K user Ram and 16K of Video Ram.

Some MSX manufacturers include the Video Ram when quoting the size of their computer whilst some manufacturers exclude the Video Ram. In this book we will include the Video Ram and we will examine machines of two sizes namely 32K Ram and 80K Ram.

BITS

A bit is the smallest fraction of the computers memory. Bits can be considered as switches which can be either ON (set) or OFF (not set or reset). When a bit is set then it contains a 1 whilst reset bits contain a 0. The 80K machines contain 917504 BITS and the 32K machines contain 524288 BITS. Since bits are a very small unit they are grouped together in bunches of 8 bits — each bunch is known as a BYTE.

BYTES

A BYTE is the smallest, directly addressable, unit of the computers memory. The 80K machines have 65536 bytes of continuous memory with addresses 0 to 65535. In the 32K machines the address range is the same except that no memory is provided between addresses 32768 and 49151. All machines are provided with a separate bank of 16384 bytes of VIDEO memory which has the address range 0 to 16383.

Each byte consists of 8 bits each of which can represent either 1 or 0. The computer uses the BINARY NUMBER SYSTEM for its internal computations and so it sees the contents of a byte as an 8 digit BINARY number. Such a number can range between 0 and 255. The significance of the value of a particular byte depends on:

a) The value.

b) The position of the byte in memory.

c) The way in which the byte is read.

To understand this please switch on your computer and do the following exercise.

EXERCISE 2

Note that the exercise should be carried out with the computer in DIRECT mode (ie. type in without line numbers so that execution is immediate).

type POKE 50000,122 — press ENTER

The computer places the number 122 into byte 50000 and then returns to command mode with the report OK. Now we are going to examine the contents of this byte in a number of different ways.

type PRINT PEEK(50000) — press ENTER

The computer displays the number 122 on screen as you would expect.

now type PRINT CHR$(PEEK(50000)) — press ENTER

This time a z is printed because you have told the computer to consider the value in address 50000 as a character.

Now try the following:

1) PRINT BIN$(PEEK(50000)) — press ENTER — BINARY NUMBER
2) PRINT HEX$(PEEK(50000)) — press ENTER — HEX NUMBER
3) PRINT OCT$(PEEK(50000)) — press ENTER — OCTAL NUMBER

Finally try this little experiment:

type in 10 REM MSX — press ENTER

This is a small basic program — LIST it to make sure that it is there.

now type POKE 32773,130 — press ENTER (80K machines)
or type POKE 49157,130 — press ENTER (32K machines)

LIST that basic program again and notice that the line has changed to:

10 MSX

The reason for this is simply that the computer expects the value in address 32773 to represent the first basic keyword in the basic program. 130 is the TOKEN for the keyword FOR — more about basic program layout and tokens later.

RANDOM ACCESS MEMORY (RAM)

In EXERCISE 2 we used the basic instructions POKE and PEEK to change or examine the contents of a byte. You can PEEK (read) the contents of a RAM byte and you can POKE (change) the contents of a RAM byte. The RAM address range on your MSX computer is as follows:

From 32768 to 65535 — 80K machines

From 49152 to 65535 — 32K machines

With all machines the area from 62336 to 65535 is reserved for SYSTEM VARIABLES and WORK AREAS - you can PEEK in this area with safety but you should only POKE if you understand the effects of your action. The system area is fully explained later in the book.

READ ONLY MEMORY (ROM)

You can PEEK any ROM byte but POKING in the ROM area has no effect. The ROM contains the BASIC language and all the ROUTINES to control the computer, the screen, the cassette, sound etc. The ROM, which is written in Z80 MACHINE CODE, contains many useful routines (ROM ROUTINES) which can be used by the machine code programmer in his own programs.

VIDEO MEMORY

The MSX range of computers are equipped with a TMS 9918A VIDEO DISPLAY PROCESSOR which handles the video display. This chip has a dedicated RAM of its own — the video RAM contains 16384 memory bytes for picture display, sprite handling, etc. The user may directly access the VRAM using VPEEK and VPOKE. The video chip and the video ram are examined in more detail later.

KILOBYTES

A byte is a small memory unit and so it is convenient to define and use a larger unit — the KILOBYTE (KB). The KB does not contain 1000 bytes as you might expect — 1 KB contains 1024 bytes. The reason for this is that the computer uses the BINARY number system and controls memory in BLOCKS of 256 bytes each — so there are 4 blocks to the KB, and 64 * 4 = 256 blocks in the main memory area.

CENTRAL PROCESSING UNIT

The CENTRAL PROCESSING UNIT (CPU) is the microchip which controls all the functions of the computer. The MSX computers use the Z80A chip as a CPU.

The Z80A has an instruction set which comprises 245 simple instructions. These instructions can be used in combinations to give about 700 instructions in all. Direct instructions to the CPU are given in MACHINE CODE which is the only "LANGUAGE" that is understandable to the CPU. The MSX machine code is Z80 machine code.

Basic program instructions are translated into machine code, by the routines in the ROM, before they can be executed by the CPU. This translation takes time and so basic programs generally run much slower than machine code programs.

INPUT/OUTPUT PORTS

Input/Output ports are used by the computer for communication with external devices such as the VDU SCREEN, THE CASSETTE, THE LINE PRINTER, THE DISC DRIVE ETC.

The Z80A CPU controls 256 INPUT and 256 OUTPUT ports — only a few of these ports are used to control the standard MSX devices. In this book you will find many useful routines which use the I/0 ports and you will learn how to use the ports which control the VDU screen.

PROGRAMMABLE SOUND GENERATOR

The MSX computers are fitted with a GENERAL INSTRUMENT PSG chip AY—3—8910. This chip is capable of producing music from 3 channels (3 notes at one time) as well as generating sound effects from the noise channels. Music can be programmed using a basic music macro language — the chip can also be accessed using the basic SOUND command. The sound capability of your MSX is discussed later in the book.

In the next chapter we will examine the MSX memory map.

CHAPTER 3

MEMORY MAPS AND SIGNPOSTS

The MSX micros are based on the Z80A microprocessor chip — this chip is an 8 bit processor but is provided with a 16 bit addressing facility. This means that the Z80A can control 65536 memory bytes within the address range 0 to 65535 — (a computer uses more than 16 bits for addresses greater than 65535).

To overcome this limitation the MSX computers use a technique known as bank switching — "parcels" of memory are switched in and out as required so that the CPU only sees 64K of memory at any one time.

SLOTS

In the MSX system the memory chips are contained in areas known as SLOTS. Each MSX computer will support 4 slots and each slot may be expanded to contain a further 4 slots. The fully expanded machine would therefore contain 16 slots.

Lets pause a moment to examine the slot concept — in this context a slot does not mean an external slot such as the cartridge connection. The 4 slots are internal to the computer and some of the slots may be connected to one or more external plug assemblies.

Each memory slot can contain 64 K of memory — this memory can be ROM or RAM. The fully expanded MSX machine can therefore contain 1024 K bytes of memory in addition to the 16 K of Video Ram.

PAGES

Figure 3.0 shows the basic slot layout. A memory slot is divided into 4 memory pages each containing 16 K bytes of memory. At any one time the CPU can handle 4 memory pages (i.e. page 0, page 1, page 2 and page 3) which can be selected from any combination of slots.

PAGE SELECTION

Page selection is controlled by Input/Output port &HA8.

Switch on your computer and type in the following:

```
A$=BIN$(INP(&HA8)): A$=STRING$(8—LEN(A$),48)
+A$:PRINT A$
```

Now press ENTER and the screen will display an 8 bit binary number — now we will look at what this number means.

Lets assume that you have a SVI 728 or SVI 738 computer — the binary number in the port &HA8 would be 01010000.

Now divide the number into four pairs of digits:

Page 3	Page 2	Page 1	Page 0
0 1	0 1	0 0	0 0

Each pair of digits is a two bit binary number which can represent a number between 0 and 3. This number is the slot number from which the respective memory page is selected.

The number 01010000 tells us that the system ROM is located in page 0 and page 1 of slot 0. The system RAM is located in page 2 and page 3 of slot 1.

Note that the MSX ROM is always located in page 0 and page 1 of slot 0 but RAM memory may be stored in any slot.

Note also that the memory configuration may be changed by outputting a different number to port &HA8 — beware of switching in basic, because you may loose control of the machine and have to switch off.

Figures 3.1 to 3.3 shows the installed memory configurations of three 80 K MSX machines.

BASIC RAM ORGANISATION

Figure 3.4 shows the layout of the BASIC RAM. The most important part of the RAM is the MACHINE SPACE at the top end of the memory. In this area the computer keeps all the

SYSTEM VARIABLES (eg. screen and character colors, cursor position, screen mode etc.), FUNCTION KEY DEFINITIONS, VARIOUS BUFFERS and all the other information that is needed for proper operation of the computer. The machine area is fully explained in chapter 8.

BOUNDARY ADDRESSES

The location of each area of the RAM is defined by the boundary addresses of that area — see figure 3.4 (eg. the ARRAY TABLE is located from ARRAY TABLE START to ARRAY TABLE END).

The boundary addresses are 16 bit addresses and each address is contained in the RAM machine area (SYSTEM VARIABLES). To read these addresses you must type a command with the following general format into your computer.

Z$ = HEX$ (PEEK(X) + 256 * PEEK(Y)) ...... HEX ADDRESS

or Z = PEEK(X) + 256 * PEEK(Y) ...... DECIMAL ADDRESS

After execution of the command the variable Z (or Z$) will contain the desired address. X and Y are of course different for each area and you must substitute the correct X and Y values.

NOTE that the computer stores 16 bit addresses or numbers in the order low byte followed by high byte (the bytes are therefore in "reverse order") and so it is necessary to multiply the second byte by 256 to read the whole number.

In the next few chapters we examine the different areas of the RAM in some detail.

FIGURE 3.0

MSX SLOT LAYOUT

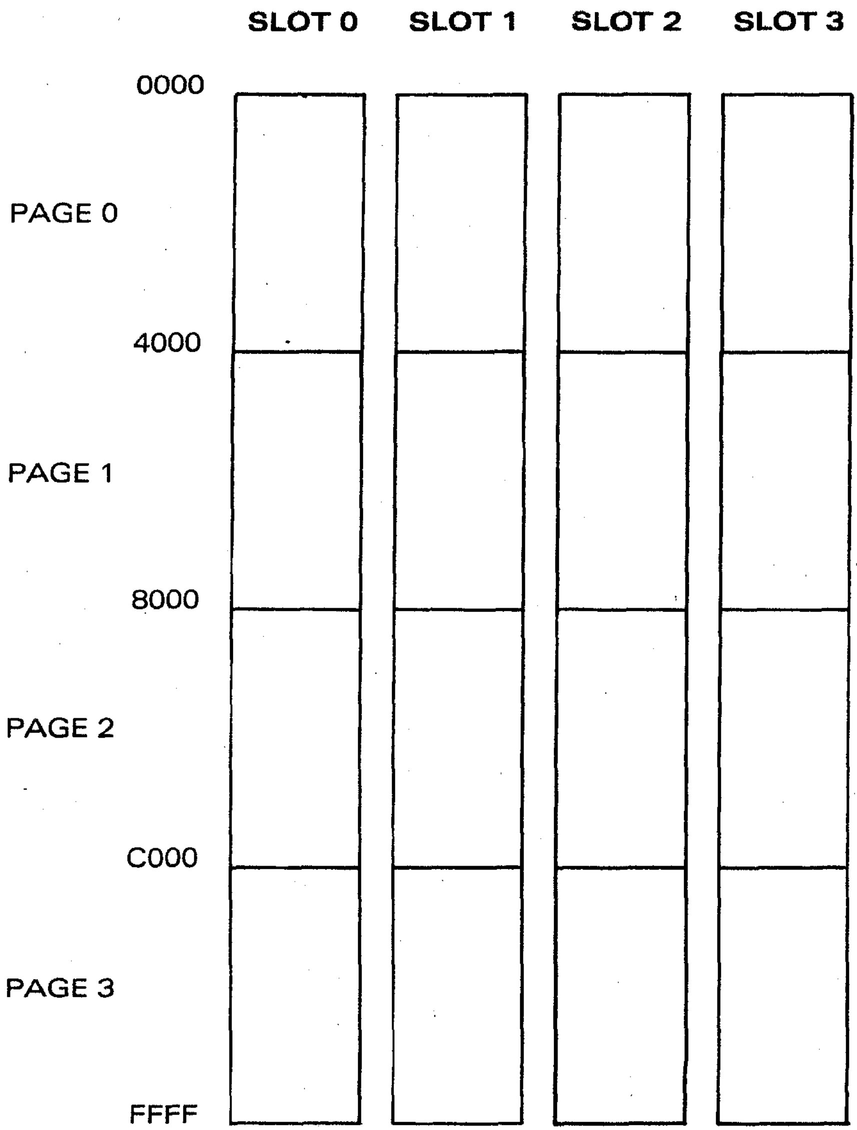


FIGURE 3.1

SVI 728/SVI 738 MEMORY LAYOUT

		SLOT 0	SLOT 1	SLOT 2	SLOT 3
	0000				
PAGE 0		ROM	CPM and MSX DOS RAM		
	4000				
PAGE 1		ROM	CPM and MSX DOS RAM		
	8000				
PAGE 2			BASIC RAM		
	C000				
PAGE 3			BASIC RAM		
	FFFF				

FIGURE 3.3

CANNON V—20 MEMORY LAYOUT

		SLOT 0	SLOT 1	SLOT 2	SLOT 3
0000					
	PAGE 0	ROM			CPM and MSX DOS RAM
4000					
	PAGE 1	ROM			CPM and MSX DOS RAM
8000					
	PAGE 2				BASIC RAM
C000					
	PAGE 3				BASIC RAM
FFFF					

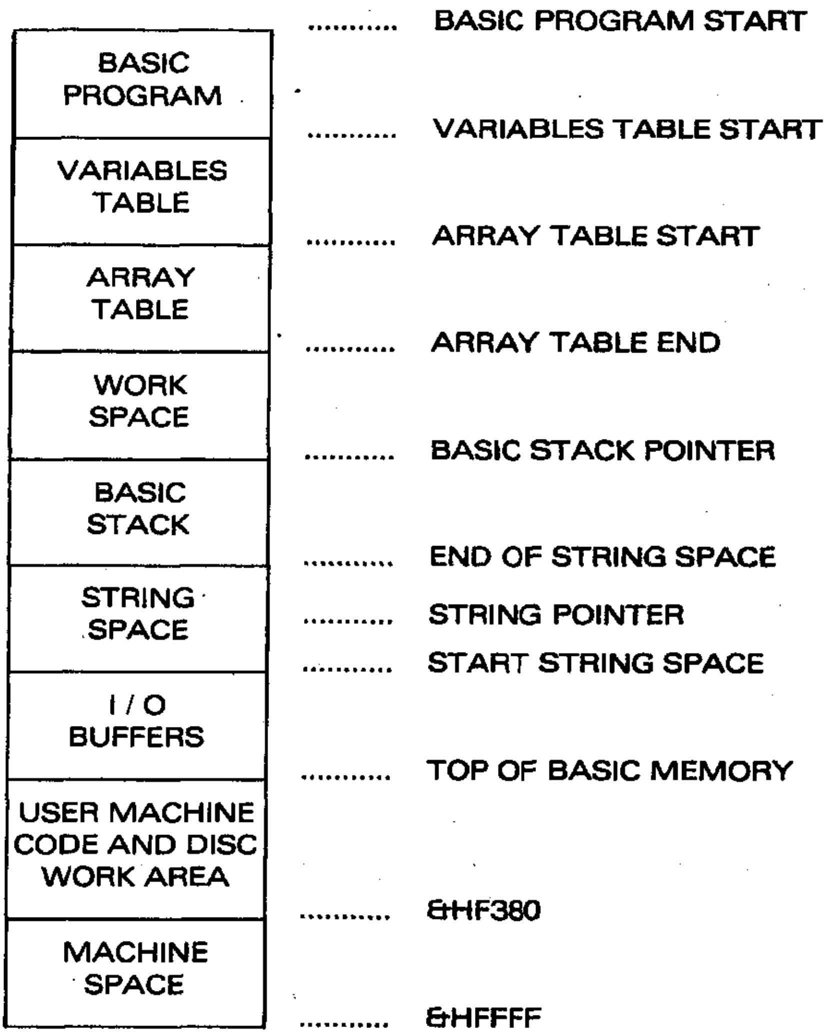
FIGURE 3.3

MITSUBISHI ML—F80 MEMORY LAYOUT

Address	Page	SLOT 0	SLOT 1	SLOT 2	SLOT 3
0000	PAGE 0	ROM	CPM and MSX DOS RAM		
4000	PAGE 1	ROM	CPM and MSX DOS RAM		
8000	PAGE 2		BASIC RAM		
C000–FFFF	PAGE 3		BASIC RAM		

FIGURE 3.4

RANDOM ACCESS MEMORY MAP UNDER BASIC



CHAPTER 4

BASIC PROGRAM AREA

The start address of the basic program area depends on the computer model — with the 80 K machines this area starts at address &H8000 whilst the basic program area of the unexpanded 32 K machine starts at &HC000.

BASIC PROGRAM START ...... X = &HF676 Y = &HF677

The basic program area is variable in length depending on the size of the program. The area ends at the start of the variables table.

VARIABLES TABLE START ...... X = &HF6C2 Y = &HF6C3

BASIC PROGRAM LAYOUT

The basic program is held within the computer in CONDENSED BINARY FORM. This means that basic KEYWORDS are held as one or two byte tokens, numbers are held in binary form and text is held in ASCII code. Each line has a memory overhead of 5 bytes which are used as shown in Table 4.1 — NOTE that "byte number" refers to the byte's position in any particular basic program line.

TABLE 4.1

byte number	1	2	3	4	....	last
contents	start address of next line		line number		....	zero

Addresses and line numbers always occupy two bytes each and so no extra memory is gained by only using small line numbers. The final byte of each line always contains a zero to indicate the end of line.

HIDING PROGRAM LINES

You may find it useful to "HIDE" certain lines or parts of lines in your program — there is a simple procedure which allows you to do this. Such lines will not appear in the program list although they remain in the program.

Program list 4.1 shows an example of the line hiding procedure to protect a password within a program. Note that the procedure replaces the Z's (in the REM statements) with DELETE marks. When the program is listed the hidden lines are printed and immediately erased — this happens so quickly that the hidden line cannot be read. You can put this routine at the start of your own programs.

Type in the program and CSAVE to the data recorder. Now RUN the program and then LIST — notice that only line 130 remains in the list — line 120 and 140 are still in the program but are hidden in the list. Now you can type in your own program from line 150 onwards. When someone RUNS your program it will erase if the user does not type in the correct password.

PROGRAM LIST 4.1

HIDDEN LINES ROUTINE

```
 10 ' hidden lines routine
 20 '
 30 ' by B L BURKE
 40 '
 50 X1=PEEK(&HF676)+256*PEEK(&HF677)
 60 X2=PEEK(&HF6C2)+256*PEEK(&HF6C3)
 70 FORY=X1TOX2
 80 IFPEEK(Y)=143ANDPEEK(Y+1)=90THENC=1:NEXT
 90 IFC=1ANDPEEK(Y)=90THENPOKEY,127ELSEC=0
100 NEXT
110 DELETE10—110
120 A$="EXCALIBER":REMZZZZZZZZZZZZZZZZZZZZZZZZ
130 CLS:INPUT"ENTER PASSWORD ";B$
140 IFB$<>A$THENNEW:REMZZZZZZZZZZZZZZZZZZZZZZZZZZZ
```

PROGRAMS WHICH MODIFY THEMSELVES

The line hiding routine is an example of a program which modifies itself – another example is given in Program list 4.2. This routine can be used in your programs so that the user can personalise the program so that it addresses him by name.

The % characters in the DATA statement are replaced with the users name and then the routine is deleted – the user must then CSAVE the personalised program. Again you can start any of your programs with a similar routine.

PROGRAM LIST 4.2

PERSONALISATION ROUTINE

```
 10 ' personalisation routine
 20 '
 30 ' by B L BURKE
 40 '
 50 CLS:INPUT"PLEASE ENTER YOUR NAME ";A$
 60 IFLEN(A$)> 20THENA$=LEFT$(A$,20)
 70 X1=PEEK(&HF676)+256*PEEK(&HF677)
 80 X2=PEEK(&HF6C2)+256*PEEK(&HF6C3)
 90 FORY=X1TOX2
100 IFPEEK(Y)=132ANDPEEK(Y+1)=34THENY=Y+1:
    GOTO110ELSENEXTY
110 FORZ=1TOLEN(A$):POKEY+Z,ASC(MID$(A$,Z,1)):
    NEXTZ
120 CLS:PRINT"PROGRAM PERSONALISATION COM-
    PLETE"
130 PRINT"PLEASE SAVE PROGRAM"
140 DELETE10-140
150 DATA"%%%%%%%%%%%%%%%%%%%%"
160 READN$:Z=INSTR(N$,"%"):N$=LEFT$(N$,Z-1)
170 CLS:LOCATE(40-6-LEN(N$))/2,2:PRINT"HELLO ";N$
```

ANIMALS

Animals is an interactive program which has been written for many different computers. My version is presented in Program list 4.3.

The user must think of an animal and the computer has to guess the animal based on the answer to a simple question. The program must be CSAVED every time the game is played because the computer learns new facts at each game. The program in its present form can accommodate 20 animals.

Enjoy the ANIMALS program — it is another example of a program which modifies itself.

NOTE the use of the system variable DATA POINTER in Program list 4.3 line 280. This variable always points to the next set of data in the basic program.

DATA POINTER ...... X = &HF6C8   Y = &HF6C9

PROGRAM LIST 4.3 — ANIMALS

```
 10 ONSTOPGOSUB210:STOPON
 20 KEY OFF
 30 CLS:LOCATE0,10:PRINT"THINK OF AN ANIMAL AND
    DONT TELL ME"
 40 LOCATE0,17:PRINT"PRESS ANY KEY ";
 50 IFINKEY$<>""THEN50
 60 IFINKEY$=""THEN60
 70 CLS:LOCATE0,10:PRINT"TELL ME A FEATURE OF
    THIS ANIMAL"
 80 LOCATE0,14
 90 INPUTAA$
100 X=0:RESTORE370
110 READF$,A$
120 P=INSTR(F$,"@"):F$=LEFT$(F$,P-1)
130 P=INSTR(A$,"@"):A$=LEFT$(A$,P-1)
140 IFF$=""THEN240
150 IFAA$=F$THENCLS:LOCATE0,10:PRINT"THE
    ANIMAL IS ";A$:GOTO170
160 X=X+1:IFX<20THEN110ELSE220
170 LOCATE0,17:PRINT"PRESS ANY KEY ";
180 IFINKEY$<>""THEN180
190 IFINKEY$=""THEN190
200 GOTO30
210 RETURN220
220 CLS:LOCATE0,10:PRINT"I AM TIRED OF GUESSING
    ANIMALS"
230 KEY ON:END
```

PROGRAM LIST 4.3 CONTINUED

```
240 CLS:LOCATE0,10:PRINT"I DONT KNOW THAT ONE"
250 PRINT:PRINT:PRINT"PLEASE TELL ME THE ANIMAL"
260 INPUT AB$
270 IFLEN(AB$)>19THENAB$=LEFT$(AB$,19)
280 DP=PEEK(&HF6C8)+256*PEEK(&HF6C9)
290 DP=DP-52
300 PE=PEEK(&HF6C2)+256*PEEK(&HF6C3)
310 FORPC=DPTOPE:IFPEEK(PC)=132ANDPEEK(PC+1)=
    64THEN320ELSENEXTPC
320 FORPK=1TOLEN(AA$)
330 POKEPC+PK,ASC(MID$(AA$,PK,1))
340 NEXTPK
350 IFAA$<>AB$THENAA$=AB$:GOTO280
360 GOTO30
370 DATA@@@@@@@@@@@@@@@@@@@@
380 DATA@@@@@@@@@@@@@@@@@@@@
390 DATA@@@@@@@@@@@@@@@@@@@@
400 DATA@@@@@@@@@@@@@@@@@@@@
410 DATA@@@@@@@@@@@@@@@@@@@@
420 DATA@@@@@@@@@@@@@@@@@@@@
430 DATA@@@@@@@@@@@@@@@@@@@@
440 DATA@@@@@@@@@@@@@@@@@@@@
450 DATA@@@@@@@@@@@@@@@@@@@@
460 DATA@@@@@@@@@@@@@@@@@@@@
470 DATA@@@@@@@@@@@@@@@@@@@@
480 DATA@@@@@@@@@@@@@@@@@@@@
490 DATA@@@@@@@@@@@@@@@@@@@@
500 DATA@@@@@@@@@@@@@@@@@@@@
510 DATA@@@@@@@@@@@@@@@@@@@@
520 DATA@@@@@@@@@@@@@@@@@@@@
530 DATA@@@@@@@@@@@@@@@@@@@@
540 DATA@@@@@@@@@@@@@@@@@@@@
550 DATA@@@@@@@@@@@@@@@@@@@@
560 DATA@@@@@@@@@@@@@@@@@@@@
570 DATA@@@@@@@@@@@@@@@@@@@@
580 DATA@@@@@@@@@@@@@@@@@@@@
590 DATA@@@@@@@@@@@@@@@@@@@@
600 DATA@@@@@@@@@@@@@@@@@@@@
610 DATA@@@@@@@@@@@@@@@@@@@@
620 DATA@@@@@@@@@@@@@@@@@@@@
```

PROGRAM LIST 4.3 CONTINUED

```
630 DATA@@@@@@@@@@@@@@@@@@@@
640 DATA@@@@@@@@@@@@@@@@@@@@
650 DATA@@@@@@@@@@@@@@@@@@@@
660 DATA@@@@@@@@@@@@@@@@@@@@
670 DATA@@@@@@@@@@@@@@@@@@@@
680 DATA@@@@@@@@@@@@@@@@@@@@
690 DATA@@@@@@@@@@@@@@@@@@@@
700 DATA@@@@@@@@@@@@@@@@@@@@
710 DATA@@@@@@@@@@@@@@@@@@@@
720 DATA@@@@@@@@@@@@@@@@@@@@
730 DATA@@@@@@@@@@@@@@@@@@@@
740 DATA@@@@@@@@@@@@@@@@@@@@
750 DATA@@@@@@@@@@@@@@@@@@@@
760 DATA@@@@@@@@@@@@@@@@@@@@
```

BASIC KEYWORDS AND TOKENS

The MICROSOFT BASIC LANGUAGE of the MSX computers has 125 basic commands/operators and 48 basic functions. The difference between a command and a function is as follows:

a) A command tells the computer to DO SOMETHING eg. PRINT, MOTOR ON etc.

b) A function tells the computer to perform some operation upon data and to RETURN A RESULT eg. X = INT(23.456) ...... returns X=23.

Commands are held as single byte tokens in a basic program and functions are held as two byte tokens. This is very memory efficient — take the word LOCATE which is often used to position the cursor for printing — the token for LOCATE is 216 ie. one byte instead of 6 for the full word LOCATE. The full basic word list and token table is given overleaf.

MSX BASIC WORD AND TOKEN TABLE

AUTO	169	AND	246	ATTR$	233
BSAVE	208	BLOAD	207	BEEP	192
BASE	201	CALL	202	CLOSE	180
COPY	214	CONT	153	CLEAR	146
CLOAD	155	CSAVE	154	CRSLIN	232
CIRCLE	188	COLOR	189	CLS	159
CMD	215	DELETE	168	DATA	132
DIM	134	DEFSTR	171	DEFINT	172
DEFSNG	173	DEFDBL	174	DSKO$	209
DEF	151	DSKI$	234	DRAW	190
ELSE	161	END	129	ERASE	165
ERROR	166	ERL	225	ERR	226
EQV	249	FOR	130	FIELD	177
FILES	183	FN	222	GOTO	137
GO TO	137	GOSUB	141	GET	178
INPUT	133	IF	139	INSTR	229
IMP	250	INKEY$	236	IPL	213
KILL	212	KEY	204	LPRINT	157
LLIST	158	LET	136	LOCATE	216
LINE	175	LOAD	181	LSET	184
LIST	147	LFILES	187	MOTOR	206
MERGE	182	MOD	251	MAX	205
NEXT	131	NAME	211	NEW	148
NOT	224	OPEN	176	OUT	156
ON	149	OR	247	OFF	235
PRINT	145	PUT	179	POKE	152
PSET	194	PRESET	195	POINT	237
PAINT	191	PLAY	193	RETURN	142
READ	135	RUN	138	RESTORE	140
REM	143	RESUME	167	RSET	185
RENUM	170	SCREEN	197	SPRITE	199
STOP	144	SWAP	164	SET	210
SAVE	186	SPC(	223	STEP	220
STRING$	227	SOUND	196	THEN	218
TRON	162	TROFF	163	TAB	219
TO	217	TIME	203	USING	228
USR	221	VARPTR	231	VPOKE	198
VDP	200	WIDTH	160	WAIT	150
XOR	248	>	238	=	239
<	240	+	241	−	242
*	243	/	244	^	245
\	252	'	230		

BASIC FUNCTIONS WORD AND TOKEN TABLE

NOTE ALL THE TOKENS IN THIS TABLE ARE PREFIXED WITH 255

ABS	134	ATN	142	ASC	149
BIN$	157	CINT	158	CSNG	159
CDBL	160	CVI	168	CVS	169
CVD	170	COS	140	CHR$	150
DSKF	166	EXP	139	EOF	171
FRE	143	FIX	161	FPOS	167
HEX$	155	INT	133	INP	144
LPOS	156	LOG	138	LOC	172
LEN	146	LEFT$	129	LOF	173
MKI$	174	MKS$	175	MKD$	176
MID$	131	OCT$	154	POS	145
PEEK	151	PDL	164	PAD	165
RIGHT$	130	RND	136	SGN	132
SQR	135	SIN	137	STR$	147
SPACE$	153	STICK	162	STRIG	163
TAN	141	VAL	148	VPEEK	152

NOTES

i) You do not type tokens in to your basic program — you type in keywords and the tokens are automatically stored in the memory instead of the characters of the keyword.

ii) Later in the book I will show you how to make use of basic tokens in your machine code programs.

iii) Using a single quote in your program instead of REM takes up 3 bytes instead of 1. The single quote tokenises as a colon, REM and the token 230, although you only see the single quote in the program list.

iv) The basic word ELSE tokenises as a colon followed by 161. The colon is not displayed in the program list.

CHAPTER 5

VARIABLES AND ARRAYS

Variables are small MEMORY BOXES which you can define to hold various numbers (NUMERIC VARIABLES) or text (STRING VARIABLES) for use in your programs. Each variable must be given a name of one or two characters (eg. A or XY etc.) and a value. You may enter your variables as follows:

LET A = 12345 or A = 12345 ...... NUMERIC VARIABLE.

LET A$ = "abc" or A$ = "abc" ...... STRING VARIABLE.

Variables like these are known as SIMPLE VARIABLES — one variable value for each variable name. Variables which have not been given a value are equal to zero in the case of numeric variables and in the case of string variables they are equal to ("") — an empty string.

The computer keeps all information about variables in a memory area called the variables table.

VARIABLES TABLE

See Figure 3.4 for the relative location of the variables table in the computer memory map. The memory area starting at the variables table is controlled by the basic system and the instructions within the basic program. The variables table contains entries for each simple variable defined in the basic program. The values of numeric variables are held within the table but in the case of string variables only the string descriptor is held in the table. Note that if you STOP a program the variables table remains intact until you change, add or re-enter a program line or RUN the program.

The variables table is located in the memory just after the basic program.

VARIABLES TABLE START ...... X = &HF6C2 Y = &HF6C3

NUMERIC VARIABLES

The space taken up by a variable depends on the precision of the number concerned:

1) DOUBLE PRECISION — Double precision variable names should be suffixed with the # sign eg. X# = 12345678901234. You can omit this sign if you define the variable using the DEFDBL instruction eg. DEFDBLX. Double precision variables can hold a number correct to 13 decimal places with the restriction that the maximum number of digits is 14. Numbers with more than 14 digits are rounded and presented in exponential form. Double precision numbers take up 11 bytes in the variables table:
   a) The first byte contains 8 to indicate double precision.
   b) Next there are two bytes for the variable name.
   c) Finally there are 8 bytes to contain the value.

2) SINGLE PRECISION — single precision variable names should be suffixed with the ! sign eg. X! = 1234567. You can omit the sign if you define the variable with a DEFSNG instruction eg. DEFSNGX. Single precision numbers are correct to 6 figures with larger numbers being rounded. Numbers with more than 14 digits are presented in exponential form. Single precision numbers take up 7 bytes in the variable table:
   a) The first byte contains 4 to indicate single precision.
   b) Next there are 2 bytes to contain the variable name.
   c) The last 4 bytes contain the variable value.

3) INTEGER VARIABLES — These variables have names which are suffixed with the % sign eg. X% = 23456. You may omit the sign if you define the variable using the DEFINT instruction eg. DEFINTX. Integer variables take up 5 bytes in the variables table and they can range in value from —32768 to 32767.

   Five byte variables table entry:
   a) The first byte contains 2 to indicate an integer.
   b) The next 2 bytes contain the variable name.
   c) The last 2 bytes contain the integer value.

STRING VARIABLES

String variables are suffixed with the $ sign eg. X$ = "asdfg". You may omit the $ sign if you define the variable using the DEFSTR instruction eg. DEFSTRX. The variables table only contains a 6 byte string descriptor for each string variable.

Six byte string descriptor:

a) The first byte contains 3 to indicate a string variable.
b) The next 2 bytes contain the variable name.
c) The next byte contains a number to indicate the number of characters in the string.
d) The last 2 bytes contain the start address of the memory area where the string is located.

NOTES

i) The variables X, X#,X!, X% and X$ are all different and can all be used in a program at the same time.
ii) The length of the variables table will change depending on the number and type of variables defined by the basic program — the table ends at the address where the ARRAY TABLE starts:

ARRAY TABLE START ...... X = &HF6C4 Y = &HF6C5

ARRAYS

Arrays are collections of variables all bearing the same name and containing similar or related data. Different ELEMENTS of the array are identified by a system of number subscripts eg. A(1) , A(2) etc.

A single DIMENSION array (vector) is a single column of numbers or strings. A table of numbers is represented by a two dimension array eg. the array A(2,2) is a numeric table with 3 columns and 3 rows.

Arrays must be properly dimensioned before you can use them. Dimension your array as follows:

DIM A(21,20) ...... numeric array with 22 rows & 21 cols.

DIMA$(10) ...... string array with 1 col & 11 rows.

ARRAY TABLE

See Figure 3.4 for the relative location of the array table in the computer memory map. ARRAYS are subscripted variables with definitions and type signs the same as for simple variables. The array table is of variable length depending on the number and magnitude of the array dimensions. The table layout is given below:

a) The first byte in an array descriptor contains a number to indicate the nature of the array variable:
   - i) 8 for double precision.
   - ii) 4 for single precision.
   - iii) 2 for integer.
   - iv) 3 for string.

b) The next 2 bytes contain the variable name (1 or 2 characters).

c) The fourth and fifth bytes contain the number of bytes remaining in the array descriptor.

d) The sixth byte contains the number of array dimensions.

e) Next there are a number of 2 byte entries one for each of the dimensions — each 2 byte entry contains the size of the relevant dimension. NOTE that in arrays the element 0 is significant so the array A(1,1) has 4 elements namely A(0,0) , A(0,1) , A(1,0) and A(1,1).

f) Finally there are entries for each element of the array as follows:
   - i) 8 byte entries for double precision arrays.
   - ii) 4 byte entries for single precision arrays.
   - iii) 2 byte entries for integer arrays.
   - iv) 3 byte string descriptors for string arrays.

The array table ends at the address stored in the system variable array table end.

ARRAY TABLE END ...... X = &HF6C6 Y = &HF6C7

****

CHAPTER 6

STRING SPACE

Refer to Figure 3.4 for the relative position of the string space within the computer memory map. Strings are collections of characters (words or sentences) which have been assigned to a string variable. Each character of the string takes up 1 byte in string space. String space starts high up in the memory and decends downwards towards the stack area. At power on your MSX computer allocates 200 bytes of string space but the user can change this using the CLEAR command.

> eg. CLEAR 2000 — allocates 2000 bytes of string space.

The useful addresses associated with string space are:

START STRING SPACE ...... X = &HF672 Y = &HF673

END STRING SPACE ...... X = &HF674 Y = &HF675

STRING POINTER ...... X = &HF69B Y = &HF69C

NOTES

i) The start of string space is dictated by the current value of the TOP OF MEMORY marker — more about that just now.

ii) The end of string space is dependent on the start address and upon the size of the string space allocated by the CLEAR command.

iii) The string pointer contains the address of the next free byte in string space.

iv) Strings can be any length up to 255 bytes long. When strings are edited (changed) there is no guarantee that the resultant string will fit into the old space allocated to that string. To overcome this problem the new version of the string is placed into string space starting at the string pointer and the string descriptor is updated to point to the new string.

v) Obsolete strings are not erased immediately but remain in memory until the string space becomes full and the computer automatically performs a garbage collection. The garbage collection consists of erasing all the obsolete strings and restacking the current strings from the start of string space. This procedure can take several minutes if a large amount of string space has been allocated.

INPUT/OUTPUT FILES

Input/Output files are used to format and control data input and output from/to various devices eg. the screen, the data recorder, the disc drive etc. The files are located in the computer memory map just above the string space and below the top of memory. At power on the computer automatically allocates space for two I/O files namely FILE 0 and FILE 1.

Addresses associated with the file space are:

TOP OF MEMORY ...... X = &HFC4A Y = &HFC4B
START OF STRING SPACE ...... X = &HF672 Y = &HF673

Up to 16 files are available on your MSX computer — you can change the number of files allocated by using the MAXFILES command:

eg. MAXFILES = 2 — allocate space for one extra file namely FILE 2.

MAXFILES = 0 — Release file space allocated to FILE 1.

Use MAXFILES = 0 if you do not need any files (eg. if your program is not doing any I/O to cassette or other device) — this will release 267 bytes for other duties. Each file uses 267 bytes — file 0 is located at the start of file space and it cannot be switched off because it is used by the computer for various automatic operations.

The actual location of any file in memory is given by:

Z = VARPTR(F) ...... where F is the file number.

Incidentally you can find the location of any variable by using the VARPTR function:

eg. PRINT VARPTR(X) ...... Z = VARPTR(A$) ...... etc.

Notice that the computer always reports a negative number as the address of a variable (VARPTR) — this is because the computer uses integers for addresses and you will recall that MSX integers range from −32768 to 32767. When the computer has to report an address which is greater than 32767 it uses the binary TWO'S COMPLEMENT FORM ie. all binary 1's become 0's and 0's become 1's, add 1 and then change the sign. To read the real address add 65536:

eg. Z = VARPTR(X) + 65536

TOP OF MEMORY

I have spoken a number of times about the TOP OF MEMORY — let's look at what this means. The memory area which is controlled by the basic system is the area between BASIC PROGRAM START and TOP OF BASIC MEMORY. If the user POKES in this area the basic program is likely to overwrite the POKED values — in order to protect such POKES it is necessary to place them above the top of basic memory. This however presents another problem because the area above the top of basic memory is reserved for SYSTEM VARIABLES and other machine controlled parameters. To get around this problem the user can lower the top of basic memory to release a space for special POKES and machine code routines. This area is shown in Figure 3.4 as USER MACHINE CODE AND DISC SYSTEM.

To reserve space above the top of memory proceed as follows:

CLEAR A,B ...... where A is the amount of STRING SPACE required and B is the required top of memory address.

BASIC HINTS

At the start of your BASIC programme you should have the various memory reconfiguration commands in the proper sequence — of course you will not always need all the commands in every program.

eg.
```
10 MAXFILES = 2
20 CLEAR 500,56000
30 DEFINTA−Z
40 DEFSTRY
50 DIMY(200)
```

The sequence is important because MAXFILES and CLEAR wipe out several other commands.

TRANSFERING DATA TO TAPE

Files are used to transfer DATA to tape. A tape DATA FILE is different from a program file in that it does not have LINE NUMBERS and it is saved in ASCII MODE ie. each character is saved as an ASCII code.

The program code required to create a data file is in the following example:

```
10 A$ = "THIS IS A DATA FILE TEST"
20 OPEN "CAS:TEST" FOR OUTPUT AS#1
30 PRINT#1,A$
40 CLOSE
```

To read the file back into your program use the following code:

```
50 OPEN "CAS:TEST" FOR INPUT AS#1
60 INPUT#1,A$
70 CLOSE
```

Note that any variable data can be transferred to tape using basic code similar to the above.

****

In the next chapter we examine a very important area of the computers memory — THE BASIC STACK.

CHAPTER 7

THE BASIC STACK

See Figure 3.4 for the relative position of the basic stack in the computers memory map. The basic stack occupies the memory area between the end of the string space and the stack pointer. The stack pointer marks the current top of the stack which grows downwards from the end of string space.

This may seem a bit of an anomaly or an error but it is quite true — the stack grows downwards and so the top of the stack is at the lowest stack memory address.

The stack is used by the basic command GOSUB and by FOR NEXT loops. Stacks work on a LAST IN FIRST OUT basis (LIFO) and items which are left on the stack will simply remain there. In extreme cases the memory can fill up due to poor stack management.

Type into your computer the following program line:

```
10 GOSUB 10
```

Now type RUN and press ENTER — notice how quickly the computer memory fills up. Each GOSUB puts a 7 byte return address onto the stack and this address is only removed when the RETURN command is executed. It is therefore essential that each GOSUB in your program is matched by a RETURN.

FOR-NEXT loops use up 25 bytes of stack space which is only cleared when the loop has run through all its cycles. It is often necessary to jump out of FOR-NEXT loops when a desired condition has been met — this practice leaves the 25 bytes on the stack. To avoid problems you should ensure that all FOR-NEXT loops are contained in sub-routines. The RETURN after the sub-routine wipes the return address and the FOR-NEXT loop off the stack.

The mini programs 7.1 and 7.2 illustrate the stack operation. Line 10 sets up the user defined function FNSP(X) as a measure of the stack pointer. The programs then print the current value of the stack pointer before and after a GOSUB — notice that the stack pointer address has reduced by 7 because the return address is now on the stack. The programs then enter the FOR-NEXT loop and again print the stack pointer address — this time the pointer has reduced by 25 because the FOR-NEXT loop is on the stack. The programs now loop until Z = 10 whilst printing the stack pointer at each loop. When the condition (Z = 10) is met program 7.1 exits the for/next loop and prints the final stack pointer address whilst program 7.2 exits the loop and returns before printing the final stack pointer.

PROGRAM LIST 7.1

```
 10 DEF FNSP(X) = PEEK(X) + 256 * PEEK(X+1)
 20 X = &HF6B1
 30 PRINT FNSP(X)
 40 GOSUB 70
 50 PRINT FNSP(X)
 60 END
 70 PRINT FNSP(X)
 80 FOR Z = 1 TO 100
 90 PRINT FNSP(X)
100 IF Z < 10 THEN NEXT
110 PRINT FNSP(X)
```

In program 7.1 there is no RETURN to match the GOSUB in line 40 and so the FOR-NEXT loop and the return address remain on the stack. Notice the final value of the stack pointer is still 32 less than the first stack pointer address. This is poor stack management.

PROGRAM LIST 7.2

```
 10 DEF FNSP(X) = PEEK(X) + 256 * PEEK(X+1)
 20 X = &HF6B1
 30 PRINT FNSP(X)
 40 GOSUB 70
 50 PRINT FNSP(X)
 60 END
 70 PRINT FNSP(X)
 80 FOR Z = 1 TO 100
 90 PRINT FNSP(X)
100 IF Z < 10 THEN NEXT
110 RETURN
```

In program 7.2 good stack management is illustrated — the program returns after exiting the FOR-NEXT loop and the stack is returned to its original condition. Notice that the final stack pointer address is equal to the first address.

The computer automatically looks after the stack but good programming (eg. list 7.2) will prevent the dreaded OUT OF MEMORY message from appearing on your screen due to poor stack management.

****

The next chapter concludes our examination of the computer memory map with an exposition of the mysteries of the machine systems area above the top of memory.

CHAPTER 8

MACHINE SYSTEMS AREA

The machine systems area contains all the SYSTEM VARIABLES which are needed for the computer to function properly. Things like the cursor position, the softkey definitions, screen colors, keyboard buffer, etc. etc. etc. .....

This chapter explores the use and position of the useful sections of the system area which is located as follows:

SYSTEMS AREA START ADDRESS = 62336 decimal or F380 hex.

SYSTEMS AREA END ADDRESS = 65535 decimal or FFFF hex.

TABLES

START ADDRESS	NAME	NO OF BYTES	DESCRIPTION
&HF39A	USR TABLE	20	10 * 2 BYTE USR ADDRESSES SET UP BY DEFUSR STATEMENT
&HF6CA	DEF TABLE	26	26 * 1 BYTE ENTRIES GIVING THE DEFAULT VARIABLE TYPE 2 = INTEGER 3 = STRING 4 = SINGLE 8 = DOUBLE CHANGE BY DEFINT ETC.
&HF87F	FUNCTION STRING TABLE	160	10 * 16 BYTE ENTRIES ONE FOR EACH FUNCTION KEY CONTAINS CURRENT STRINGS
&HF975	MUSIC A	128	MUSIC QUEUE USED BY PLAY
&HF9F5	MUSIC B	128	MUSIC QUEUE USED BY PLAY

TABLES CONTINUED

START ADDRESS	NAME	NO OF BYTES	DESCRIPTION
&HFA75	MUSIC C	128	MUSIC QUEUE USED BY PLAY
&HFB41	VOICE A	36	STATIC DATA FOR MUSIC A
&HFB66	VOICE B	36	STATIC DATA FOR MUSIC B
&HFB8B	VOICE C	36	STATIC DATA FOR MUSIC C
&HFBCE	FUNCTION FLAGS	10	INDICATES IF FUNCTION KEY TRAP IS ON = 1 OR OFF = 0
&HFC4C	TRAP TABLE	30	10 * 3 BYTE ENTRIES FOR F—KEY TRAPS BYTE 1 OFF = 0 ON = 1 BYTES 2/3 ADDRESS OF TRAP GOSUB LINE
&HFC6A	STOP TRAP	3	BYTE 1 OFF = 0 ON = 1 BYTES 2/3 ADDRESS OF TRAP GOSUB LINE
&HFC6D	SPRITE TRAP	3	BYTE 1 OFF = 0 ON = 1 BYTES 2/3 ADDRESS OF TRAP GOSUB LINE
&HFC70	STRIG TRAPS	5*3	BYTE 1 OFF = 0 ON = 1 BYTES 2/3 ADDRESS OF TRAP GOSUB LINE
&HFC7F	INTERVAL TRAP	3	BYTE 1 OFF = 0 ON = 1 BYTES 2/3 ADDRESS OF TRAP GOSUB LINE
&HFC82	DEVICE TRAP TABLE	8*3	TO TRAP EVENTS FROM EXTERNAL DEVICES e.g. RS232
&HF3B3	TEXT	10	5*2 BYTE BASE ADDRESSES FOR VDP IN TEXT MODE

TABLES CONTINUED

START ADDRESS	NAME	NO OF BYTES	DESCRIPTION
&HF3BD	32 COL	10	5*2 BYTE BASE ADDRESSES FOR VDP IN SCREEN 1
&HF3C7	GRAPHICS	10	5*2 BYTE BASE ADDRESSES FOR VDP IN SCREEN 2
&HF3D1	LO RES	10	5*2 BYTE BASE ADDRESSES FOR VDP IN SCREEN 3
&HFD9A	HOOK JUMP TABLE	560	112*5 BYTE HOOKS USED TO HOOK YOUR OWN ROUTINES INTO BASIC ROM ROUTINES

USEFUL PARTS OF THE MUSIC STATIC DATA TABLE

BYTE NO	ENTRY FUNCTION
3	LENGTH OF MUSIC STRING
4 — 5	ADDRESS OF M STRING
11 — 12	TONE PERIOD
13	AMPLITUDE/SHAPE
14 — 15	ENVELOPE PERIOD
16	OCTAVE
17	NOTE LENGTH
18	TEMPO
19	VOLUME

USEFUL ADDRESSES

TO READ THE ACTUAL ADDRESS USE:

Z = PEEK(LOW BYTE) + 256 * PEEK(HIGH BYTE)

LOW BYTE	HIGH BYTE	ADDRESS NAME
&HF674	&HF675	END OF STRING SPACE
&HF676	&HF677	BASIC PROGRAM START
&HF672	&HF673	START OF STRING SPACE
&HF69B	&HF69C	STRING POINTER
&HF6A1	&HF6A2	POINTER TO END OF FOR LOOP
&HF6AF	&HF6B0	RESUME ADDRESS
&HF6B1	&HF6B2	STACK POINTER ADDRESS
&HF6B3	&HF6B4	LAST ERROR LINE NUMBER
&HF6B5	&HF6B6	CURRENT LINE USED BY LIST.
&HF6B9	&HF6BA	ERROR HANDLING LINE NUMBER
&HF6BE	&HF6BF	LAST LINE WHEN CTRL/STOP
&HF6C0	&HF6C1	RESTART ADDRESS USED BY CONT
&HF6C2	&HF6C3	START OF VARIABLES TABLE
&HF6C4	&HF6C5	START OF ARRAY TABLE
&HF6C6	&HF6C7	END OF ARRAY TABLE
&HF6C8	&HF6C9	ADDRESS OF NEXT DATA
&HF862	&HF863	ADDRESS OF FILE#0 BUFFER
&HF3F8	&HF3F9	END POINTER IN KEY BUFFER
&HF3FA	&HF3FB	START POINTER IN KEY BUFFER
&HFC4A	&HFC4B	TOP OF BASIC MEMORY
&HFC9E	&HFC9F	COUNTER FROM 0 TO 65535
&HFCA0	&HFCA1	CURRENT INTERVAL VALUE
&HFCA2	&HFCA3	INTERVAL COUNT DOWN

MORE SYSTEM VARIABLES AND FLAGS

ADDRESS	CONTENTS	POKE
&HF414	LATEST ERROR NUMBER	NO
&HF3B0	SCREEN LINE LENGTH	YES
&HF6AA	AUTO LINE NUMBERING FLAG 1 = ON 0 = OFF	YES
&HF85F	NUMBER OF FILES – MAXFILES	NO
&HF3DB	CLICK SWITCH 1 = ON 0 = OFF	YES
&HF3DC	CURSOR LINE	NO
&HF3DD	CURSOR COLUMN	NO
&HF3DE	FUNCTION KEY DISPLAY SWITCH	NO
&HF3E9	FOREGROUND COLOR	YES
&HF3EA	BACKGROUND COLOR	YES
&HF3EB	BORDER COLOR	YES

NOTE THAT POKED COLORS ONLY BECOME ACTIVE AFTER A SCREEN INSTRUCTION.

EVEN MORE SYSTEM VARIABLES AND FLAGS

ADDRESS	CONTENTS	POKE
&HFCAB	UPPER CASE CHARACTERS FLAG 1 = ON 0 = OFF	YES
&HFCAF	SCREEN MODE NUMBER	NO

VDP REGISTERS

ADDRESS	CONTENTS
&HF3DF	REGISTER 0 OF VDP
&HF3E0	REGISTER 1 OF VDP
&HF3E1	REGISTER 2 OF VDP
&HF3E2	REGISTER 3 OF VDP
&HF3E3	REGISTER 4 OF VDP
&HF3E4	REGISTER 5 OF VDP
&HF3E5	REGISTER 6 OF VDP
&HF3E6	REGISTER 7 OF VDP
&HF3E7	REGISTER 8 OF VDP

AUTO RUN PROGRAM

Here is a short program which uses the systems area to AUTORUN a CLOAD program.

PROGRAM LIST 8.1

```
10 FORX=0TO3:READZ:POKE&HFBF0+X,Z:NEXT
20 POKE&HF3FA,&HF0:POKE&HF3FB,&HFB
30 POKE&HF3F8,&HF4:POKE&HF3F9,&HFB
40 DATA82,85,78,13
50 CLOAD
```

Type in the program and SAVE it to tape in ASCII MODE ie. you must SAVE the program and not CSAVE. Save the program with the following command:

SAVE "AUTO"

Now CSAVE your own program onto the tape just after the AUTO program. When you wish to RUN your program you type in:

RUN "AUTO"

The program AUTO will load and run and after loading your program it will automatically RUN.

AUTO works as follows:

1) The word RUN followed by the ENTER code is poked into the key buffer.

2) The key buffer pointers are reset to point to the start and end of the instruction RUN.

3) Your program is then CLOADED.

4) After loading is complete the computer returns to command mode and the instruction RUN is ejected from the key buffer and immediately executes.

In the next few chapters we take a detailed look at the Video Chip.

CHAPTER 9

THE VIDEO CHIP

The MSX computers use the TMS 9918A video chip to handle all screen output. This chip has 4 different screen modes all of which are implemented on the MSX machines.

In this chapter we take a look at the way the video chip works.

GENERAL

The MSX picture is made up of 35 different planes stacked one on top of the other. These planes are numbered from 0 to 34 with plane 34 being at the bottom of the pile. Images on the lower planes can only be seen if the upper planes are transparent at that particular point.

The lowest plane of all is plane 34 — this is the external video plane. The use of this plane (to display pictures from an external video chip or other video source) is not implemented on most MSX machines.

Immediately above the external video plane is the backdrop plane which is a single color plane and cannot display any images. This plane provides the border around the graphics screens.

The next plane is the pattern plane (in screen 3 this is the multicolor plane). This plane displays all the pattern images created with PRINT, DRAW, LINE, CIRCLE etc. etc.

All the remaining planes (31 — 0) are for sprites — one sprite can be displayed on each plane making a total of 32 sprites displayed at one time. Sprites on the upper planes (lower plane numbers) will pass in front of sprites on the lower planes. Only four sprites may be displayed in any horizontal line — the fifth sprite in a line will disappear.

CONTROL

The MSX machines are provided with a dedicated bank of 16K bytes of video RAM. The video chip controls the display by maintaining a series of tables in the video RAM memory. The tables are set up differently for each of the four display modes — screen 0, screen 1, screen 2 and screen 3. Different modes are set up using the nine registers of the video chip.

VDP TABLES

1) PATTERN GENERATOR — The pattern generator table is an area of video ram which contains the data for producing shapes on the pattern plane. The data is held in binary 8 bit numbers so that when displayed the binary 1 will produce a dot in the foreground color and binary 0 will remain in the background color. Patterns are formed by grouping 8 binary numbers together to form a pattern block. The character set definitions are held in a pattern generator table.

2) COLOR TABLE — The color table is similar to the pattern generator table except that the data refer to foreground and background colors rather than display positions.

3) SPRITE PATTERN GENERATOR TABLE — Same as the pattern generator table except that the patterns refer to sprites which can be displayed on the sprite planes. Sprite patterns are defined in blocks of 8 binary numbers — large sprites are formed from 4 such blocks.

4) NAME TABLE — For the purposes of the name table the screen is divided up into small squares and the name table has an entry for each square. These entries define which pattern block is to be displayed in that particular square.

5) SPRITE ATTRIBUTE TABLE — The sprite attribute table has 32 * 4 byte entries one entry for each of the sprite display planes. An entry consists of:

   a) Y co-ordinate — display position down the screen.

   b) X co-ordinate — display position across the screen.

   c) Sprite number — 0 to 255 for 8 * 8 sprites.
   0 to 255 step 4 for 16 * 16 sprites.

   d) Sprite color — Bits 0 to 3 contain the color.
   Bit 7 is used to move the sprite to the left in order to facilitate entry from behind the left border.

VIDEO CHIP REGISTERS

In order to set up and maintain control over the various tables the video chip has a set of 9 registers.

REGISTER 0

BIT NO.	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	X	E

Only two bits of register 0 are used namely bit 0 and bit 1. Bit 0 is the EXTERNAL VIDEO ENABLE BIT which is normally set to zero on the MSX machines. Bit 1 is marked X and will be discussed under register 1.

****

REGISTER 1

BIT NO.	7	6	5	4	3	2	1	0
	R	B	I	Y	Z	0	S	M

All bits are significant with the exception of bit 2 which is reserved for future expansion.

BIT 0 — The sprite magnification bit —
0 for normal size.
1 for double size.

BIT 1 — The sprite size bit — 0 for 8 * 8 sprites.
1 for 16 * 16 sprites.

BIT 2 — Reserved.

BIT 3, BIT 4 AND BIT 1 REGISTER 0 — These 3 bits act together as in the following table.

	X	Y	Z
TEXT SCREEN 0	0	1	0
TEXT SCREEN 1	0	0	0
GRAPHICS SCREEN 2	1	0	0
GRAPHICS SCREEN 3	0	0	1

BIT 5 — The VDP interrupt enable bit — 0 to disable interrupt.
1 to enable interrupt.

BIT 6 — The video enable bit — 0 to disable the display.
1 to enable the display.

BIT 7 — The RAM select bit — 0 to select a 4K video RAM.
1 to select a 16K video RAM.

****

REGISTER 2

BIT NO.	7	6	5	4	3	2	1	0
	0	0	0	0	name table base address			

Register 2 contains a number between 0 and 15 from which the BASE ADDRESS of the name table can be calculated.

NAME TABLE BASE ADDRESS = (REGISTER 2) * 400 HEX

****

REGISTER 3

BIT NO.	7	6	5	4	3	2	1	0
	color table base address							

Register 3 contains a number between 0 and 255 — The COLOR TABLE BASE ADDRESS is calculated as follows:

COLOR TABLE BASE ADDRESS = (REGISTER 3) * 40 HEX

****

REGISTER 4

BIT NO.	7	6	5	4	3	2	1	0
	0	0	0	0	0	pattern generator		

Register 4 contains a number between 0 and 7 from which the PATTERN GENERATOR BASE ADDRESS can be calculated.

PATTERN GENERATOR BASE ADDRESS =
(REGISTER 4) * 800 HEX

REGISTER 5

BIT NO.	7	6	5	4	3	2	1	0
	0	sprite attribute table base address						

Register 5 contains a number between 0 and 127 which defines the SPRITE ATTRIBUTE TABLE position in the video RAM.

SPRITE ATTRIBUTE TABLE BASE ADDRESS =
(REGISTER 5) * 80 HEX

****

REGISTER 6

BIT NO.	7	6	5	4	3	2	1	0
	0	0	0	0	0	sprite pattern		

Register 6 contains a number in the range 0 to 7 from which the SPRITE PATTERN GENERATOR BASE ADDRESS can be calculated.

SPRITE PATTERN GENERATOR BASE ADDRESS =
(REGISTER 6) * 800 HEX

****

REGISTER 7

BIT NO.	7	6	5	4	3	2	1	0
	text color				back drop color			

Register 7 controls the global colors. In text mode the backdrop section of the register contains the background color whilst in graphics mode register contains the border color.

****

REGISTER 8

BIT NO.	7	6	5	4	3	2	1	0
	F	S	C	fifth sprite plane number				

The interpretation of Register 8 is as follows:

BIT 7 — This is the interrupt flag which is set to 1 every time the VDP completes a screen scan.

BIT 6 — This is the FIFTH SPRITE FLAG and is set to 1 whenever there are five sprites in a horizontal line across the screen. When five sprites are in a horizontal line across the screen then the sprite on the lowest plane (highest plane number) will disappear.

BIT 5 — This is the sprite coincidence flag which is set to 1 whenever two sprites collide.

BITS 0 TO 4 — These bits contain the plane number of the fifth sprite.

****

This concludes the examination of the VIDEO CHIP — in the next few chapters you will learn how to use the VDP and its registers in some advanced ways.

CHAPTER 10

DIRECT ACCESS TO THE VIDEO CHIP AND VIDEO RAM

The MSX communicates with the VDP and the VRAM through 4 INPUT/OUTPUT PORTS. The ports concerned are as follows:

1) OUTPUT PORT &H98 ..... WRITE VRAM DATA.
2) OUTPUT PORT &H99 ..... WRITE ADDRESS OR REGISTER NUMBER
3) INPUT PORT &H98 ..... READ VRAM DATA.
4) INPUT PORT &H99 ..... RESET STATUS REGISTER.

WRITING TO THE VDP REGISTERS

1) Decide on the data to the output to the register and place the data into variable X. eg. X = 19.
2) Decide on the register to which the data is to be output and place the register number into variable Y. eg. Y = 7.
3) Output the data in the following way:

```
10 Z=INP(&H99)
20 OUT&H99,X
30 OUT&H99,(YOR&H80)
```

Type RUN followed by ENTER to transfer the data to the VDP register.

NOTE

a) Z = INP(&H99) resets the STATUS REGISTER to enable a good transfer to take place.
b) In line 30 data bit 7 is set (ie. register number OR &H80 is output) to signal to the VDP that we wish to transfer data to a register and not to VRAM memory.
c) There is an easier way (in Basic) to write to a VDP register using the basic word VDP.
e.g. VDP(7) = 19.

READING FROM THE VIDEO RAM MEMORY

1) Decide on the VRAM address from which you want to start reading data — place this address into variable X — eg. X = 275.
2) Convert the address into a 4 digit hex number = &H0113.

3) Divide the hex address into a low byte = &H13 and a high byte = &H01.

4) Reset the status register.

5) Send the low byte out through port &H99.

6) Send the high byte out through port &H99.

7) Read the data in through port &H98.

SPECIAL NOTE TO PROGRAMMERS

In order to maintain compatibility with all future versions of MSX, the port numbers to read and write VDP memory data have been placed at set addresses in ROM.

VDP read data port number is contained in address &H0006.

VDP write data port number is contained in address &H0007.

Your program should read the relevant port number from these addresses before performing a direct read or write operation. Future versions of MSX may use different ports but your program will still operate correctly because the port numbers were read from the ROM.

PROGRAM LIST 10.1

VRAM DIRECT READ

```
 10 ' example VRAM direct read
 20 CLS:DI=PEEK(6)
 30 VPOKE&H113,65
 40 VPOKE&H114,78
 50 X=INP(&H99)
 60 OUT&H99,&H13
 70 OUT&H99,&H1
 80 Z1=INP(DI)
 90 Z2=INP(DI)
100 PRINTZ1,Z2
```

Program 10.1 illustrates the method of direct reading of the video RAM. Two characters are poked onto the screen and then the VRAM address is output through port &H99. The character codes are then read directly through the port contained in variable DI — NOTE that the VRAM address increments automatically after every read.

WRITING TO THE VIDEO RAM MEMORY

1) Decide on the VRAM address to which you want to start writing data — place this address into variable X — eg. X = 275.

2) Convert the address into a 4 digit hex number = &H0113.

3) Divide the hex address into a low byte = &H13 and a high byte = &H01.

4) Reset the status register.

5) Send the low byte out through port &H99.

6) Send the high byte or &H40 out through port &H99. NOTE that bit 6 is set to inform the video chip that we want to do a VRAM WRITE OPERATION.

7) Write the data out through the output port indicated in ROM byte 7.

PROGRAM LIST 10.2

VRAM DIRECT WRITE

```
10 'example VRAM direct write
20 CLS:DO=PEEK(7)
30 Z1=65
40 Z2=78
50 X=INP(&H99)
60 OUT&H99,&H13
70 OUT&H99,(&H1OR&H40)
80 OUTDO,Z1
90 OUTDO,Z2
```

This program illustrates the method of directly writing to the video RAM memory. Two ASCII codes are placed in variables Z1 and Z2. The VRAM destination address is output through port &H99 — first the low byte and then the high byte or &H40. The two data bytes are then output through the port contained in variable DO. NOTE that the destination address automatically increments with each write operation.

CHAPTER 11

TEXT MODE

The MSX text mode is known as SCREEN 0 – this is the default mode which is always current when the computer is switched on. (NB. MSX produced for the Japanese market defaults to screen 1).

TEXT MODE VDP REGISTER CONTENTS

REGISTER 0 = 0
REGISTER 1 = &HF0
REGISTER 2 = 0 .... NAME TABLE STARTS AT 0.
REGISTER 3 = ? .... NOT SIGNIFICANT IN TEXT MODE.
REGISTER 4 = &H1 .... PATTERN GENERATOR STARTS AT &H800.
REGISTER 5 = ? .... NOT SIGNIFICANT IN TEXT MODE.
REGISTER 6 = ? .... NOT SIGNIFICANT IN TEXT MODE.
REGISTER 7 = &HF4 .... WHITE TEXT/BLUE BACKGROUND.
REGISTER 8 = ? .... DEPENDS ON INTERRUPT STATUS.

NOTES

1) In text mode the screen is divided into 960 pattern positions each of which is capable of displaying a character. There are 40 positions in each row and 24 rows.

2) The pattern NAME TABLE starts at VRAM address 0 as defined by (register 2) * &H400 = 0 * &H400 = 0.

3) Each entry in the name table represents a pattern position on the screen. Position 0 is in the top left of the screen. The position numbers increase across the screen so that the top right hand position is 39 and the second row ranges from 40 on the left to 79 on the right. Position mapping is illustrated in figure 11.1.

4) There is a one to one correspondence between the screen character position and the character code position in the name table. eg. The character in screen position 167 is contained in VRAM byte 167 which is the 168'th entry in the name table.

FIGURE 11.1

TEXT SCREEN CHARACTER POSITION MAP

0	1	2	....	37	38	39
40	41	42	....	77	78	79
80	81	82	....	117	118	119
120	121	122	....	157	158	159
160	161	162	....	197	198	199
200	201	202	....	237	238	239
240	241	242	....	277	278	279
280	281	282	....	317	318	319
320	321	322	....	357	358	359
360	361	362	....	397	398	399
400	401	402	....	437	438	439
440	441	442	....	477	478	479
480	481	482	....	517	518	519
520	521	522	....	557	558	559
560	561	562	....	597	598	599
600	601	602	....	637	638	639
640	641	642	....	677	678	679
680	681	682	....	717	718	719
720	721	722	....	757	758	759
760	761	762	....	797	798	799
800	801	802	....	837	838	839
840	841	842	....	877	878	879
880	881	882	....	917	918	919
920	921	922	....	957	958	959

PATTERN GENERATOR TABLE

In text mode the pattern generator table contains the character set and is located at (register 4) * &H800 = 1 * 2048 decimal — ie. the character set starts at VRAM address 2048. Each character is defined in an 8 byte block of VRAM memory and the maximum number of character definitions in the generator table is 256.

Characters are defined as follows:

CHARACTER A:	2568	00100000	32
	2569	01010000	80
	2570	10001000	136
	2571	10001000	136
	2572	11111000	248
	2573	10001000	136
	2574	10001000	136
	2575	00000000	0
CHARACTER S:	2712	01110000	112
	2713	10001000	136
	2714	10000000	128
	2715	01110000	112
	2716	00001000	8
	2717	10001000	136
	2718	01110000	112
	2719	00000000	0

Program list 11.1 is a short program to display all the character definitions on the screen. Interpret the display as follows:

a) The number on the left is the VRAM address of the byte containing the relevant piece of character data.

b) The data is displayed in binary form in the middle of the screen and in decimal form on the right of the screen. You may change any character by using VPOKE to change the character data.

c) Notice that the two least significant bits are always zero for the ASCII characters — this is because the standard MSX character is defined in a 6*8 block of dots.

PROGRAM LIST 11.1

```
10 ' character definitions
20 SCREEN 0:FORX=2048TO4097STEP8
30 FORY=0TO7
40 B$=BIN$(VPEEK(X+Y))
50 B$=STRING$(8-LEN(B$),48)+B$
60 PRINTX+Y,B$;TAB(28)VAL("&B"+B$)
70 NEXTY
80 PRINT:PRINT
90 NEXTX
```

****

CHARACTER SETS

The video chip will support up to 7 different character sets held in video memory at the same time. The sets must be located starting at an 800 hex address boundary and the set currently in use is selected using the VDP register 4.

CHARACTER SET DEFINITION TABLES START ADDRESSES

SET NUMBER	VRAM START ADDRESS	REGISTER 4
1	2048	1
2	4096	2
3	6144	3
4	8192	4
5	10240	5
6	12288	6
7	14336	7

Set 1 is the standard character set to which the MSX defaults at power on. The other 6 sets must be user defined or constructed by modifying set 1. The following 3 program lists (11.2, 11.3, 11.4) demonstrate the use of the other character sets. In each of the programs a second character set is created by modifying the standard set — call the new set using GOTO 100 and return to the standard set using GOTO 200.

NOTE that you must be in screen 0 when running these programs.

PROGRAM LIST 11.2

THE INVERSE SET

```
 10 ' inverse set located as set 7
 20 FORX=0TO2047
 30 VPOKE 14336+X,255-VPEEK(2048+X)
 40 NEXT
 99 ' call inverse set
100 VDP(4)=7
110 END
199 ' restore normal set
200 VDP(4)=1
210 END
```

****

PROGRAM LIST 11.3

THE UNDERLINE SET

```
 10 ' underline set located as set 2
 20 FORX=0TO2047
 30 VPOKE4096+X,VPEEK(2048+X)
 40 NEXT
 50 FORX=15TO2047STEP8
 60 VPOKE4096+X,255
 70 NEXT
 99 ' call underline set
100 VDP(4)=2
110 END
199 ' restore normal set
200 VDP(4)=1
210 END
```

PROGRAM LIST 11.4

THE UPSIDEDOWN SET

```
 10 ' upsidedown set located as set 3
 20 FORX=0TO2047STEP8
 30 FORY=7TO0STEP-1
 40 VPOKE6144+X+7-Y,VPEEK(2048+X+Y)
 50 NEXTY
 60 NEXTX
 99 ' call upsidedown set
100 VDP(4)=3
110 END
199 ' restore normal set
200 VDP(4)=1
210 END
```

****

USING REGISTER 7

In text mode register 7 defines the foreground (ink) and background (paper) colors. It works like this:

1) Select the foreground color — eg. black = 1.

2) Select the background color — eg. yellow = 11.

3) Convert the color numbers into HEX — foreground = 1.
   background = B.

4) Join the two hex numbers together = 1B.

5) Output this value to register 7 using the following:

   ```
   10 VDP(7)=&H1B
   ```

   Now type RUN followed by ENTER and the colors will change to BLACK TEXT on a YELLOW BACKGROUND.

   That concludes the examination of the TEXT MODE — in the next chapter we look at the VDP in 32 column text mode (screen 1).

CHAPTER 12

THE 32 COLUMN TEXT SCREEN

The MSX 32 column text screen (SCREEN 1) provides a text screen which can display 15 colors plus transparent — sprites can also be used on this screen. The screen uses the GRAPHICS 1 mode of the TMS 9918A video chip which will support 256 two color characters or user defined graphics (8 * 8 dot picture blocks).

32 COLUMNS TEXT VDP REGISTER CONTENTS

REGISTER 0 = &H00
REGISTER 1 = &HE0
REGISTER 2 = &H06 .... NAME TABLE BASE ADDRESS
= &H1800
REGISTER 3 = &H80 .... COLOR TABLE BASE ADDRESS
&H2000
REGISTER 4 = &H00 .... PATTERN GEN BASE ADDRESS
&H0000
REGISTER 5 = &H36 .... SPRITE ATTRIBUTE TABLE
&H1B00
REGISTER 6 = &H07 .... SPRITE PATTERN TABLE
&H3800
REGISTER 7 = ? .... DEPENDS ON THE BORDER COLOR
REGISTER 8 = ? .... DEPENDS ON INTERRUPT STATUS

NOTES

1) Imagine the screen is divided up into 768 blocks and each block consists of 8 * 8 dots or PIXELS (picture elements). There are 32 blocks in a row and 24 rows on the screen.

2) The PATERN GENERATOR table is located from 0 to &H7FF in the video ram — the table is filled with the standard MSX character set when screen 1 is first selected. Any or all of the characters may be redefined by inserting data bytes using VPOKE.

3) The NAME TABLE has 768 entries one for each picture block on the screen. When SCREEN 1 is blank then each entry in the name table is 32 which corresponds to the ASCII value of the space character. The entry of any ASCII value in the name table will result in the corresponding character being displayed on the screen.

4) The COLOR TABLE has 32 entries each entry defining a unique forground and background color for a block of 8 characters or UDG.

To illustrate these concepts switch on your computer and type in the following mini program:

```
10 SCREEN 1
20 FOR X = 0 TO 255
30 VPOKE &H1800+X,X
40 NEXT
50 LOCATE 0,12
```

Now type RUN followed by ENTER.

This little program displays the MSX characters by placing the ASCII values into the NAME TABLE.

Now for some color — list the program and change it as follows:

```
10 SCREEN 1
20 FOR X = 0 TO 255
30 VPOKE &H1800+X,X
40 NEXT
50 VPOKE &H2000,&H1B
60 VPOKE &H2001,&H8B
70 LOCATE 0,12
```

The new lines 50 and 60 make entries in the first two positions of the color table — notice that each color table entry is in two parts the first hex digit refering to the forground color and the second digit refering to the background color. Notice also that each color entry controls the color of a block of 8 characters.

Now lets look at defining a UDG — list the program and modify as follows:

```
 10 SCREEN 1
 20 FOR X = 0 TO 255
 30 VPOKE &H1800+X,X
 40 NEXT
 50 VPOKE &H2000,&H1B
 60 VPOKE &H2001,&H8B
 70 FOR X = 0 TO 7
 80 READ Y
 90 VPOKE X,Y
100 NEXT
110 LOCATE 0,12
120 DATA 66,60,90,125
130 DATA 60,24,36,66
```

RUN the program and notice the new character (a space invader) in the top left hand corner — this character is of course defined by the data in lines 120 and 130.

Screen 1 will support three different pattern generator tables or sets of 256 characters (UDG). Only one table may be active on screen at any one time.

The tables are located as follows:

TABLE 0 VRAM ADDRESS &H0000 TO &H07FF
TABLE 1 VRAM ADDRESS &H0800 TO &H0FFF
TABLE 2 VRAM ADDRESS &H1000 TO &H17FF

Table 0 and 1 are initialised to contain the MSX character set by a SCREEN 1 call. Table 2 must be defined by the user. To select a particular table use the VDP (4) command:

eg. VDP (4) = 2 .... to select table 2.

MOVEMENT

Before moving on to the graphics screens lets try some movement on screen 1.

First run the program which defines the space invader in character position zero. Now type NEW and type in the following program:

```
10 CLS
20 FOR X = &H1801 TO &H1AFF
30 VPOKE X-1,32
40 VPOKE X,0
50 FOR Y = 1 TO 100
60 NEXT
70 NEXT
```

RUN this program — it causes our space invader to move on the screen. Note that the Y loop slows down the action and can be changed to change the speed. Note also that when UDG are moved then the image in the old position has to be erased — line 30 does this. Delete line 30 and see what happens.

REMEMBER that screen 1 supports all sprite functions this together with UDG, color and movement make screen 1 a very useful screen for games programming.

In the next chapter we look at the high resolution graphics screen 2.

CHAPTER 13

THE HIGH RESOLUTION SCREEN

The MSX high resolution screen (SCREEN 2) provides a resolution of 256 dots across the screen and 192 dots down the screen. The screen uses the GRAPHICS 2 mode of the TMS 9918A video chip which can display 15 colors plus transparent in a standard 8 * 8 dot picture block (user defined graphic).

HI—RES GRAPHICS VDP REGISTER CONTENTS

REGISTER 0 = &H02
REGISTER 1 = &HE0
REGISTER 2 = &H06 .... NAME TABLE BASE ADDRESS = &H1800
REGISTER 3 = &H80 .... COLOR TABLE BASE ADDRESS = &H2000
REGISTER 4 = &H00 .... PATTERN GEN BASE ADDRESS = &H0000
REGISTER 5 = &H36 .... SPRITE ATTRIBUTE TABLE = &H1B00
REGISTER 6 = &H07 .... SPRITE PATTERN TABLE = &H3800
REGISTER 7 = ? .... DEPENDS ON THE BORDER COLOR
REGISTER 8 = ? .... DEPENDS ON INTERRUPT STATUS

NOTES

1) Imagine the screen is divided up into 768 blocks and each block consists of 8 * 8 dots or PIXELS (picture elements). Further imagine that the screen is divided horizontaly into three equal sections — each section contains 256 picture blocks. There are 32 blocks in each line and 8 lines in each section making a total of 24 lines on the screen.

2) The NAME TABLE has three sections — one for each section of the screen. Each section of the name table has 256 entries — one for each picture block in the screen section. When SCREEN 2 is first selected the name table entries correspond to the screen positions — ie. the first entry in each section is 0 the next is 1 and so on to the last entry in the section which is 255. This means that any entry in the PATTERN GENERATOR TABLE will immedia[illegible] ecome visible on the screen.

3) Lets make an entry into the pattern generator table to illustrate these concepts:

```
10 OPEN "GRP:"FOR OUTPUT AS#1
20 SCREEN 2
30 PSET(1,0),4
40 PRINT#1,"A"
50 GOTO 50
```

This mini program appears to PRINT an "A" in the top left hand corner of the screen — in fact we have transfered the 8 pieces of data which define "A" into the first 8 entries of the pattern generator table. Further we have shifted that data one dot to the right so that the "A" is more central within the graphics 8 * 8 picture block.

The first 8 bytes in the pattern generator table now look as follows:

BYTE &H0000	00010000
BYTE &H0001	00101000
BYTE &H0002	01000100
BYTE &H0003	01000100
BYTE &H0004	01111100
BYTE &H0005	01000100
BYTE &H0006	01000100
BYTE &H0007	00000000

Now press CTRL/STOP and modify the mini program as follows:

```
10 OPEN "GRP:"FOR OUTPUT AS#1
20 SCREEN 2
30 PSET (1,0),4
40 PRINT#1,"A"
50 PSET (9,0),4
60 PRINT#1,"B"
70 GOTO 70
```

When you RUN this program we find a "B" next to the "A" on the screen — we have now transfered the 8 data bytes which define "B" into the next 8 entries of the pattern generator table. Again we have made the character more central within the 8 * 8 picture block.

The next 8 bytes in the pattern generator table now look like this:

BYTE &H0008	01111000
BYTE &H0009	00100100
BYTE &H000A	00100100
BYTE &H000B	00111000
BYTE &H000C	00100100
BYTE &H000D	00100100
BYTE &H000E	01111000
BYTE &H000F	00000000

4) Now lets introduce some color into our two user defined graphics. The color table starts at VRAM address &H2000 and there is one color entry to match each entry in the pattern generator table. Color table entries are constructed as follows:

   i) Decide on a foreground color — ie. the color to be assumed by the 1's in the pattern definition — eg. RED = 6.

   ii) Decide on a background color — ie. the color to be assumed by the 0's in the pattern definition — eg. YELLOW = 10.

   iii) Convert color numbers into hex:—
   FOREGROUND = 6,
   BACKGROUND = A

   iv) Join the two hex digits together = 6A.

   v) Place the result into the correct place in the color table:
   eg. VPOKE &H2000,&H6A

Now press CTRL/STOP and modify the mini program as follows:

```
 10 OPEN "GRP:" FOR OUTPUT AS#1
 20 SCREEN 2
 30 PSET (1,0),4
 40 PRINT#1,"A"
 50 PSET (9,0),4
 60 PRINT  #1,"B"
 70 FOR X = 0 TO 7
 80 VPOKE &H2000+X,&H6A
 90 VPOKE &H2008+X,&HA6
100 NEXT
110 GOTO 110
```

The first few entries in the color table now look like this:

```
&H2000    &H6A
  :         :
  :         :
&H2007    &H6A
&H2008    &HA6
  :         :
  :         :
&H200F    &HA6
```

NOTE that each data entry in the pattern generator table has a corresponding entry in the color table and the address of the color entry is equal to the address of the pattern generator table entry plus &H2000.

5) Lets now examine the NAME TABLE — at the moment the name table is set up so that any screen position will display its corresponding block graphic as defined in the pattern generator table. So for example screen position 0 (top left hand corner) displays the "A" which is defined in position 0 of the pattern generator table and in position 0 of the color table. Likewise the "B" is in position 1 on the screen and in the tables.

If we change the name table entries we can move the image on the screen — to illustrate this press CTRL/STOP and modify the mini program as follows:

```
 10 OPEN "GRP:" FOR OUTPUT AS#1
 20 SCREEN 2
 30 PSET (1,0),4
 40 PRINT#1,"A"
 50 PSET (9,0),4
 60 PRINT#1,"B"
 70 FOR X = 0 TO 7
 80 VPOKE &H2000+X,&H6A
 90 VPOKE &H2008+X,&HA6
100 NEXT
110 FOR X = 0 TO 255
120 VPOKE &H1800+X,0
130 NEXT
140 FOR X = 0 TO 255
150 VPOKE &H1800+X,1
160 NEXT
170 FOR X = 0 TO 255
180 VPOKE &H1800+X,2
190 NEXT
200 VPOKE &H1850,0
210 VPOKE &H18FF,1
220 GOTO 220
```

NOTES

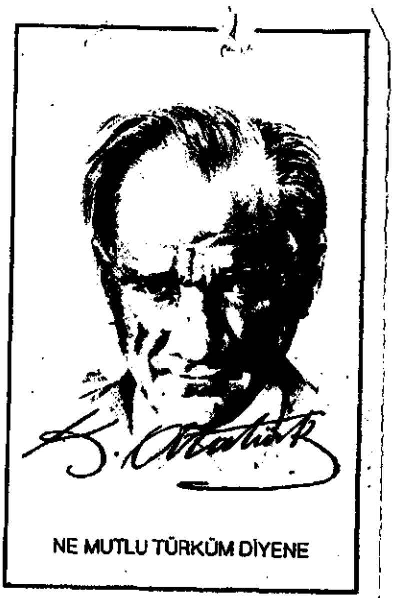
a) Lines 110 to 130 fill the top section of the screen with user defined graphic 0 — the "A".

b) Lines 140 to 160 fill the top section of the screen with user defined graphic 1 — the "B".

c) Lines 170 to 190 fill the top section of the screen with user defined graphic 2 — undefined and therefore just blank.

d) Finally lines 200 and 210 place the user defined graphics at specific locations on the screen section.

USER DEFINED GRAPHICS ARE ONLY ACTIVE IN THE SCREEN SECTION FOR WHICH THEY WERE DEFINED — YOU WILL RECALL THAT THE SCREEN IS DIVIDED INTO 3 SECTIONS — TOP THIRD, MIDDLE THIRD, AND BOTTOM THIRD. EACH SCREEN THIRD HAS ITS OWN SET OF UDG.



SCREEN 2 TABLE ADDRESSES (TOP THIRD)

NAME TABLE (TOP THIRD)

&H1800	&H1801	&H1802	....	&H181D	&H181E	&H181F
&H1820	&H1821	&H1822	....	&H183D	&H183E	&H183F
&H1840	&H1841	&H1842	....	&H185D	&H185E	&H185F
&H1860	&H1861	&H1862	....	&H187D	&H187E	&H187F
&H1880	&H1881	&H1882	....	&H189D	&H189E	&H189F
&H18A0	&H18A1	&H18A2	....	&H18BD	&H18BE	&H18BF
&H18C0	&H18C1	&H18C2	....	&H18DD	&H18DE	&H18DF
&H18E0	&H18E1	&H18E2	....	&H18FD	&H18FE	&H18FF

Each address represents the name table address for that particular screen location in the top third of the screen.

PATTERN GENERATOR AND COLOR TABLE ADDRESSES (TOP THIRD)

PATTERN GENERATOR	COLOR TABLE
&H0000	&H2000
&H0001	&H2001
&H0002	&H2002
:::::: ::::::	:::::: ::::::
&H07FD	&H27FD
&H07FE	&H27FE
&H07FF	&H27FF

NOTE that the first 8 entries in the pattern and color tables refer to user defined graphic 0, the second 8 entries refer to UDG1, and so on — the last 8 entries refer to UDG255. Note also that each UDG number is unique to the top third of the screen.

SCREEN 2 TABLE ADDRESSES (MIDDLE THIRD)

NAME TABLE (MIDDLE THIRD)

&H1900	&H1901	&H1902	....	&H191D	&H191E	&H191F
&H1920	&H1921	&H1922	....	&H193D	&H193E	&H193F
&H1940	&H1941	&H1942	....	&H195D	&H195E	&H195F
&H1960	&H1961	&H1962	....	&H197D	&H197E	&H197F
&H1980	&H1981	&H1982	....	&H199D	&H199E	&H199F
&H19A0	&H19A1	&H19A2	....	&H19BD	&H19BE	&H19BF
&H19C0	&H19C1	&H19C2	....	&H19DD	&H19DE	&H19DF
&H19E0	&H19E1	&H19E2	....	&H19FD	&H19FE	&H19FF

Each address represents the name table address for that particular screen location in the middle third of the screen.

PATTERN GENERATOR AND COLOR TABLE ADDRESSES (MIDDLE THIRD)

PATTERN GENERATOR	COLOR TABLE
&H0800	&H2800
&H0801	&H2801
&H0802	&H2802
:::::: ::::::	:::::: ::::::
&H0FFD	&H2FFD
&H0FFE	&H2FFE
&H0FFF	&H2FFF

NOTE that the first 8 entries in the pattern and color tables refer to user defined graphic 0, the second 8 entries refer to UDG1, and so on — the last 8 entries refer to UDG255. Note also that each UDG number is unique to the middle third of the screen.

SCREEN 2 TABLE ADDRESSES (BOTTOM THIRD)

NAME TABLE (BOTTOM THIRD)

&H1A00	&H1A01	&H1A02	....	&H1A1D	&H1A1E	&H1A1F
&H1A20	&H1A21	&H1A22	....	&H1A3D	&H1A3E	&H1A3F
&H1A40	&H1A41	&H1A42	....	&H1A5D	&H1A5E	&H1A5F
&H1A60	&H1A61	&H1A62	....	&H1A7D	&H1A7E	&H1A7F
&H1A80	&H1A81	&H1A82	....	&H1A9D	&H1A9E	&H1A9F
&H1AA0	&H1AA1	&H1AA2	....	&H1ABD	&H1ABE	&H1ABF
&H1AC0	&H1AC1	&H1AC2	....	&H1ADD	&H1ADE	&H1ADF
&H1AE0	&H1AE1	&H1AE2	....	&H1AFD	&H1AFE	&H1AFF

Each address represents the name table address for that particular screen location in the bottom third of the screen.

PATTERN GENERATOR AND COLOR TABLE ADDRESSES (BOTTOM THIRD)

PATTERN GENERATOR	COLOR TABLE
&H1000	&H3000
&H1001	&H3001
&H1002	&H3002
:::::: ::::::	:::::: ::::::
&H17FD	&H37FD
&H17FE	&H37FE
&H17FF	&H37FF

NOTE that the first 8 entries in the pattern and color tables refer to user defined graphic 0, the second 8 entries refer to UDG1, and so on — the last 8 entries refer to UDG255. Note also that each UDG number is unique to the bottom third of the screen.

The procedure is as follows:

1) Set the SCREEN mode and then read the current value of register 1 into a variable (X).

2) Set the VDP interrupt bit and the video enable/disable bit to zero by using the binary mask &B10011111 (&H9f) in conjunction with the bitwise AND instruction.

3) Write the new value to the VDP register 1.

4) Perform any basic instructions to draw your picture on the graphics screen.

5) Restore the old value in the VDP register 1. The picture is instantly displayed on the screen.

This procedure is illustrated in program list 13.1.

PROGRAM LIST 13.1

```
 10 COLOR15,4,4
 20 SCREEN2
 30 '
 40 ' x=vdp reg1 data
 50 '
 60 X = VDP(1)
 70 '
 80 ' disable screen
 90 '
100 Z=(X AND &H9F)
110 VDP(1)=Z
120 '
130 CIRCLE(90,90),20,8
140 PAINT(90,90),8
150 LINE(10,10)-(180,50),11,BF
```

PROGRAM LIST 13.1 (CONTINUED)

```
160 '
170 ' enable screen
180 '
190 VPD(1)=X
200 '
210 GOTO210
```

Change lines 130 to 150 in order to draw your own picture behind the scenes.

****

In the next chapter we look at the use of the VDP status register — register 8.

CHAPTER 14

THE VDP STATUS REGISTER

The VDP status register (register 8) is a read only register which can be used to indicate the following:

1) When there are 5 sprites in a line.

2) The plane number of the fifth sprite which has disappeared.

3) When two or more sprites have collided.

Program list 14.1 illustrates the use of the status register:

PROGRAM LIST 14.1

```
 10 SCREEN 2
 20 FORX=0TO7
 30 A$=A$+CHR$(255)
 40 NEXT
 50 SPRITE$(0)=A$
 60 PUTSPRITE0,(150,50),11,0
 70 PUTSPRITE1,(160,50),8,0
 80 PUTSPRITE3,(170,50),13,0
 90 PUTSPRITE4,(180,50),14,0
100 FORX=-20TO150
110 PUTSPRITE2,(X,50),3,0
120 NEXT
130 Z=VDP(8)
140 SCREEN0
150 PRINTBIN$(Z)
```

The program places 4 sprites in a line and then introduces a fifth sprite which moves from the right and collides with one of the other sprites. The screen then changes to screen 0 and the binary value of the status register is printed.

This value is &B11100100. Interpret as follows:

a) Bit 5 (third from the left) is a 1 so there has been a sprite collision.

b) Bit 6 (second from the left) is also 1 so there are 5 sprites in a line — the 5 junior bits give the sprite plane of the fifth sprite = 4. Notice that the sprite on the lowest plane (of the five) disappears.

Program 14.2 shows another use of the VDP register 8 — to detect which sprites have collided. The program works like this:

1) The sprite collision is detected by the normal ON SPRITE routine with the GOSUB set to line 200.

2) The routine at line 200 performs the following operations:

   i) Switches off each sprite in turn.

   ii) Allows time for the register 8 to be updated.

   iii) Checks if the sprite collision flag is still active.

   iv) Switches the sprite back on if flag still active.

   v) Displays plane number of collision sprite.

Note this routine can only be used to detect collisions when one of the colliding sprites is known — eg. a bullet or missile sprite.

PROGRAM LIST 14.2

```
 10 ONSTOPGOSUB290:STOPON
 20 DEFINTA-Z
 30 SCREEN1
 40 ONSPRITEGOSUB200
 50 FORX=0TO7
 60 A$=A$+CHR$(255)
 70 NEXT
 80 DEFFNSC(S)=(VDP(S)AND&B00100000)
 90 S=8
100 SPRITE$(0)=A$
110 FORP=1TO15
120 PUTSPRITEP,(50+P*10,P*10),P,0
130 NEXT
140 PD=(INT((RND(-TIME)*150)/10))*10
150 SPRITEON
160 FORZ=-20TO255
```

PROGRAM LIST 14.2 (CONTINUED)

```
170 PUTSPRITE0,(Z,PD),3,0
180 NEXT
190 GOTO140
200 SPRITEOFF
210 FORX=4TO60STEP4
220 YP=VPEEK(&H1B00+X)
230 IFYP=209THENNEXTELSEVPOKE&H1B00+X,209
240 FORWT=1TO50:NEXT
250 IFFNSC(S)=0THENPRINTX/4:RETURN140
260 VPOKE&H1B00+X,YP
270 NEXTX
280 RETURN140
290 SCREEN0
300 END
```

In the next chapter we examine the VDP in low resolution graphics mode.

CHAPTER 15

THE LOW RESOLUTION SCREEN

The MSX low resolution screen (SCREEN 3) provides a resolution of 64 squares across the screen and 48 squares down the screen. The screen uses the MULTICOLOR mode of the TMS 9918A video chip which can display 15 colors plus transparent on the screen. Full sprite facilities are also provided.

LO-RES GRAPHICS VDP REGISTER CONTENTS

```
REGISTER 0 = &H00
REGISTER 1 = &HE8
REGISTER 2 = &H02 .... NAME TABLE BASE ADDRESS
                       = &H0800
REGISTER 3 = &H00 .... COLOR TABLE BASE ADDRESS
                       = &H0000
REGISTER 4 = &H00 .... PATTERN GEN BASE ADDRESS
                       = &H0000
REGISTER 5 = &H36 .... SPRITE ATTRIBUTE TABLE
                       = &H1B00
REGISTER 6 = &H07 .... SPRITE PATTERN TABLE
                       = &H3800
REGISTER 7 = ?    .... DEPENDS ON THE BORDER COLOR
REGISTER 8 = ?    .... DEPENDS ON INTERRUPT STATUS
```

NOTES

1) Notice that the PATTERN and COLOR tables coincide on this screen. This is because patterns are generated by simpy lighting up different squares in different colors.

2) The screen is divided up into 768 blocks and each block consists of 2 * 2 squares (each square is made up of 4 pixels). There are 32 blocks in each row and 4 rows in each section. There are 6 sections and so there are a total of 24 rows on the screen. Take particular note of the difference between squares, blocks, and rows.

3) The blocks are arranged in columns with 4 blocks in a column and the columns are arranged in sections — 32 columns to the section and 6 sections of columns on the screen.

4) The pattern/color table has six sections each containing 256 entries — one entry for each pair of squares in a section of columns. Each of the entries defines the colors of two adjacent squares on the screen.

5) The NAME TABLE has six sections and each section has 128 entries — one for each picture block in a section of columns.

The table layout for the low resolution graphics screen is rather complicated but the following diagrams and discussion may help you to understand it:

FIGURE 15.1

ADDRESS LAYOUT OF PATTERN/COLOR TABLE (TOP SIXTH)

0	1	2		29	30	31
&H0000	&H0008	&H0010	....	&H00E8	&H00F0	&H00F8
&H0001	&H0009	&H0011	....	&H00E9	&H00F1	&H00F9
&H0002	&H000A	&H0012	....	&H00EA	&H00F2	&H00FA
&H0003	&H000B	&H0013	....	&H00EB	&H00F3	&H00FB
&H0004	&H000C	&H0014	....	&H00ED	&H00F4	&H00FC
&H0005	&H000D	&H0015	....	&H00EE	&H00F5	&H00FD
&H0006	&H000E	&H0016	....	&H00EF	&H00F6	&H00FE
&H0007	&H000F	&H0017	....	&H19FD	&H00F7	&H00FF

Each address represents the pattern/color address for that particular screen location in the top sixth or section of the screen.

The following table depicts the entire pattern/color table:

SECTION NUMBER	ADDRESS RANGE	COLUMN RANGE
1	&H0000 to &H00FF	0 to 31
2	&H0100 to &H01FF	32 to 63
3	&H0200 to &H02FF	64 to 95
4	&H0300 to &H03FF	96 to 127
5	&H0400 to &H04FF	128 to 159
6	&H0500 to &H05FF	160 to 191

Each address in the table contains the color definitions for two squares on the screen (remember that a square is made up of 4 pixels). The color data is best illustrated by a two digit hex number — the left hand digit defines the color of the left hand square and the right hand digit defines the color of the right hand square.

Lets examine these concepts with a little program:

```
10 SCREEN 3
20 FOR X = 0 TO 7
30 READ A$
40 VPOKE X,VAL("&H"+A$)
50 NEXT
60 GOTO 60
70 DATA 1B,2C,3D,6E
80 DATA B1,C2,D3,E6
```

Type this in your computer and RUN it.

Notice that the first column in the top row of the screen is filled with colored squares. If we put color data into VRAM byte &H0008 then the colors would appear at the top of the screen (next column).

Now lets look at the name table — the name table top section is illustrated in the following diagram:

FIGURE 15.2.

NAME TABLE ADDRESSES (TOP ROW)

1	2	3		29	30	31
&H0800	&H0801	&H0802	....	&H081D	&H081E	&H081F
&H0820	&H0821	&H0822	....	&H083D	&H083E	&H083F
&H0840	&H0841	&H0842	....	&H085D	&H085E	&H085F
&H0860	&H0861	&H0862	....	&H087D	&H087E	&H087F

Each address represents the name table address for that particular screen block in the top section of the screen.

Notice that there are half as many entries in the name table section as there were in the pattern/color table — this is because a name table entry is controlling a pattern block consisting of four squares whilst the p/c table entry is controlling only two squares.

NOTE that when screen 3 is first called each name table entry is the same as the respective column number — so entries in the name table column 0 are all 0 and entries in the name table column 96 are all 96.

Lets look at the table entries which control the column of colored squares produced by the little program.

COL,ROW	PATTERN/COLOR TABLE		NAME TABLE	
0,0	&H0000	&H1B	&H0800	&H00
	&H0001	&H2C		
0,1	&H0002	&H3D	&H0820	&H00
	&H0003	&H6E		
0,2	&H0004	&HB1	&H0840	&H00
	&H0005	&HC2		
0,3	&H0006	&HD3	&H0860	&H00
	&H0007	&HE6		

Notice that all the entries in the name table are zero — the zero refers to the first column in the top section. The video chip uses the row number (not stored in a table) and the name table entry to decide which block to display. If you vpoke a zero in the name table at say column 96 row 2 (vram address &H09C0) position then the image displayed will be identical to that at column zero row 2.

****

The layout of the VDP is difficult to explain and understand but you are urged to spend some time experimenting with the VDP facilities until you fully understand all the powerful features of the video chip.

In the next chapter we take a first look at machine code.

CHAPTER 16

MACHINE CODE

The remainder of this book is devoted to an introduction to Z80 MACHINE CODE and its implementation on the MSX computers. Programs presented include a full Z80 assembler which the reader can find on the tape supplied with this book.

WHAT IS MACHINE CODE?

The microprocessor which is the heart of your computer performs its various tasks in response to a set of instructions — these instructions are called machine code. In the case of the MSX computers the processor is the Z80A and the instruction set is known as Z80 machine code.

Machine code is the only language which is understood by the Z80 chip — high level languages such as BASIC are broken down into raw machine code by the BASIC INTERPRETER in the ROM before the Z80A chip can execute the instructions.

The machine code programmer has to break every task into simple steps as he codes a program — he is rewarded for his efforts by an enormous increase in operating speed. To illustrate the concept of breaking a task into parts consider the following example:

TASK	—	Make a cup of coffee.
PARTS	—	Go to kitchen. Find kettle. Collect kettle. Find water. Collect water in kettle. Find power point. Plug in kettle. Etc. Etc. Etc.

In computer terms the Basic (for humans but complex for the computer) instruction may be — PRINT "MSX" — but the machine code equivalent will consist of many small individual steps.

MACHINE CODE INSTRUCTIONS

In machine code the user can instruct the processor to perform various arithmetic and logical operations on data stored within the computer memory. Data transfers can also be performed within the memory and between the computer and various peripheral devices.

The machine code instructions and program data are stored in the computer memory and the current instruction is indicated by a pointer known as the PROGRAM COUNTER. To execute a given instruction the user must simply point the program counter at the memory byte containing that instruction.

MACHINE CODE AND THE MSX

When the MSX computer is operating under the standard basic language the basic system is in control of the whole memory area. Under these conditions your basic programs can easily overwrite any machine code you may place in the memory. To prevent this from happening you must reserve some space which is safe from the basic system before you install the machine code program.

Safe places for a machine code routine are:

a) In a basic REM statement.

b) In a string.

c) Above the top of basic memory.

The best place to install your machine code is above the top of memory after reserving space by lowering the top of memory. Look back at Chapter 6 to see how the CLEAR command is used to lower the top of memory before installing a machine code routine.

To execute the machine code routine it is necessary to set the PROGRAM COUNTER to point to the start address of the routine. This is done using the DEF USR command followed, by the Z = USR(0) command.

Machine code is just a series of numbers held within an area of memory — each number is part of a Z80 instruction or a piece of program data.

NOTE that machine code programs can be operated without any basic support — such programs can be recorded on tape using the BSAVE command and RUN using the BLOAD,R command.

eg. BSAVE "TEST",START ADDRESS,END ADDRESS,RUN ADDRESS
BLOAD "TEST",R.

THE Z80A CHIP

FIGURE 15.1

MAIN REGISTERS

F	A
B	C
D	E
H	L

ALTERNATE REGISTERS

F'	A'
B'	C'
D'	E'
H'	L'

16 BIT REGISTERS

IX
IY
SP
PC
R : IV

IN ADDITION TO THE ABOVE REGISTERS THE Z80A IS EQUIPPED WITH 256 INPUT PORTS AND 256 OUTPUT PORTS FOR COMMUNICATION WITH PERIPHERAL DEVICES SUCH AS THE SCREEN, TAPE, DISC ETC.

Figure 15.1 is a schematic diagram of the Z80A chip — lets now look at the method of operation.

THE REGISTERS

There are two sets of working registers labelled MAIN REGISTERS and ALTERNATE REGISTERS. The user can select any one set of A,F registers with either set of B,C,D,E,H,L registers to be active at any one time. The register set which is not currently in use may be used as storage because any data contained within those registers is retained. Each of the registers is an 8 bit register (ie. it can contain a number between 0 and 255) but under certain circumstances the registers may be used in pairs as 16 bit registers.

THE A REGISTER

This register is also known as the ACCUMULATOR and is used for most arithmetical and logical operations. The status of the A register (following such an operation) may be tested by checking the flag register. This information may then be used for various conditional jumps and calls.

THE F REGISTER

This is the FLAG register which contains various flags to indicate the condition of the A register following an arithmetic or logical operation.

THE B AND C REGISTERS

Usually used as loop counters (BC = byte counter) but can also be used for temporary storage and other operations.

THE D AND E REGISTERS

Used for general work and as the destination address pointer in block moves (DE = DEstination).

THE H AND L REGISTERS

These registers are generally used together as a 16 bit address pointer with the HIGH BYTE of the address in register H (H = High) and the LOW BYTE in the L register (L = Low).

THE IX AND IY REGISTERS

These registers are known as the INDEX REGISTERS and are used as address pointers. The actual address pointed to is calculated as the sum of the register contents and a specified offset or displacement between −128 and +127.

THE SP REGISTER

The STACK POINTER REGISTER contains the address of the current top of the stack. The programmer can set aside any area of the computer RAM memory as a stack area or use the area set aside by basic for a stack (provided that the MC program is called from basic).

All stack operations are 16 bit operations — the stack is used for RETURN addresses and can be used as a temporary storage area for register contents. The PUSH command pushes the contents of a 16 bit register (eg. HL or BC) onto the stack whilst the POP command pops the value off the stack into a 16 bit register. NOTE that registers A and F act as a 16 bit register for stack operations.

Remember that the stack grows downward in memory so the stack pointer is automatically decremented by 2 when a number is added to the stack. The pointer increments by 2 when a number is removed from the stack.

THE PC REGISTER

The PC register is the program counter which contains the address of the byte containing the current machine code instruction. The program counter is automatically incremented after each instruction is executed. The PC is changed by each JUMP, CALL or RETURN command.

THE R AND IV REGISTERS

The REFRESH and INTERRUPT VECTOR registers are used in advanced programming and can be ignored.

THE Z80A INSTRUCTION SET

Z80 machine code consists of over 700 instructions which can be grouped into 8 main groups:

1) LOAD AND EXCHANGE INSTRUCTIONS

Data can be taken from any memory byte or from any register and LOADED into another register or into any memory byte. Registers B, C, D, E, H, and L may be used individually (as 8 bit registers) or in pairs BC, DE, and HL as 16 bit registers.

The exchange instructions are used to exchange the contents of one register with the contents of another register.

2) BLOCK TRANSFER AND BLOCK SEARCH INSTRUCTIONS

Block transfer instructions transfer a specified number of bytes from one memory location to another.

Block search instructions search for a specific byte in a specified area of the computer memory.

3) LOGICAL AND ARITHMETIC INSTRUCTIONS

The logical operations AND, OR, and XOR can be performed between the A register and another register or memory byte.

Arithmetic operations include ADD, SUBTRACT, INCREMENT (increase by 1) and DECREMENT (decrease by 1).

4) ROTATE AND SHIFT INSTRUCTIONS

These instructions are used to ROTATE or SHIFT the bits within a specified register. A shift to the left effectively multiplies the register contents by 2 and a shift to the right divides by 2.

5) BIT MANIPULATION INSTRUCTIONS

These instructions allow the user to SET, RESET, or TEST a specific bit in a specified register or memory byte.

6) CALL, JUMP and RETURN INSTRUCTIONS

These instructions change the contents of the PROGRAM COUNTER so the program will continue operating from a different address. JUMP is similar to a basic GOTO, CALL is similar to a basic GOSUB and RETURN equates to a basic RETURN.

Jumps can be to a specified address or can be relative to the current address up to 127 bytes forward or 128 backward counting the displacement byte as −1.

7) INPUT/OUTPUT INSTRUCTIONS

The INPUT instructions can read a byte from any INPUT port into any of the registers. The OUTPUT instructions send a byte from any register to any OUTPUT port. There are also instructions which send or receive a block of bytes through a specified port.

8) Z80 CONTROL INSTRUCTIONS

HALT and the interrupt control instructions fall into this category.

****

MACHINE CODE MNEMONICS

To remember an instruction set which consists of over 700 sets of numbers is a formidable task and so it is fortunate that THE ZILOG CORPORATION OF CALIFORNIA (the originators of the Z80 chip) designed a set of mnemonics (memory aids) to assist the user to write in machine code. A machine code program which is written in mnemonics is known as a SOURCE FILE which is made up of SOURCE CODE.

An ASSEMBLER takes a source file and turns it into true machine code — the machine code file created by the assembler is known as the OBJECT FILE which consists of OBJECT CODE.

MACHINE CODE CONVENTIONS

THE BRACKETS RULE

A source code without brackets means that the operation specified must be carried out upon the contents of the register concerned.

eg. LD HL,dddd — means load register HL with number dddd.

DEC DE — means decrement the contents of register DE.

ADD A,B — means add the contents of register B to the contents of register A and leave the result in register A.

A source code with brackets means that the operation must be carried out on the contents of the memory byte which is pointed to by the address contained in the bracketed register.

eg. LD A,(HL) — means load the A register with the contents of the memory byte pointed to by the address held in the HL register.

DEC (HL) — means decrement the contents of the memory byte pointed to by the address held in the HL register.

LD(ADDR),A — means load the memory address contained in the brackets with the contents of the A register.

THE ORDER RULE

Where an instruction contains two registers or an address and a register the first named register or address will contain the result of the operation.

eg. LD SP,HL — the Stack Pointer is loaded with the contents of the HL register.

ADD SP,IY — add the contents of the IY register to the Stack Pointer and put the result into the Stack Pointer.

THE IMPLIED "A" RULE

Where an instruction obviously needs two registers but the mnemonic only contains one then the other register is always the ACCUMULATOR.

eg. XOR D — means XOR the D register with the A register and put the result into the A register.

SUB B — means subtract the contents of the B register from the contents of the A register and put the result into the A register.

THE 16 BIT RULE

When a 16 bit transfer takes place then the LOW BYTE is placed into the specified address and the HIGH BYTE is placed into the address + 1. This bit order applies for all 16 bit transfer operations — NOTE however that 16 bit registers contain the high byte *in the left hand portion of the register* (eg. B,D or H) and the low byte in the right hand part of the register (C,E or L).

****

The full list of machine code mnemonics is presented in the APPENDIX 1 for your convenience.

Do not worry if you dont understand machine code immediately you will understand more and more as you do the exercises presented in the next few chapters. The SUPER ASSEMBLER operating instructions are presented in the next chapter.

CHAPTER 17

THE SUPER ASSEMBLER

The SUPER ASSEMBLER is a full Z80 machine code assembler for the MSX computers. The assembler was written in machine code by my young friend BENNIE VAN DER MERWE.

NOTE: The Super Assembler will only work with machines which have at least 48 K RAM (including VRAM).

The assembler is located from address 48000 to 52480 or in HEX from &HBB80 to &HCD00. Machine code programs can be assembled at addresses above the end of the assembler but please ensure that you do not assemble in the machine area at the top of memory.

Do not assemble above 62336 (&HF380) because this would interfere with the system variables

LOADING THE ASSEMBLER

To load the assembler you simply type CLOAD "MSX" followed by ENTER to load the loader program. RUN the loader program which will then load the super assembler and initialise KEY (F1) with the "ASSEMBLE" command. NOTE that all the mini programs and source files in this book will be found on your SUPER ASSEMBLER tape.

SPECIAL NOTE

If you try to assemble when there is no source file in memory then an "unprintable error" will occur. You can only assemble properly constructed source files.

****

SOURCE FILES

Source files are located from address &H8000 or 32768 Decimal — i.e. the normal basic program position.

Source files are typed into the computer in the same way as basic programs except that each line is a REM statement. You may use all the basic editing features (AUTO, RENUM, etc.) when writing your source file. The SOURCE FILE must be organised in a special way for the assembler to work properly. Many examples of source files can be found in the remaining chapters of this book but the general rules are laid out below:

1) NUMBERS — The assembler can deal with numbers which are entered in HEX, DECIMAL, BINARY or OCTAL. It is however necessary to indicate which number system is used in every instance. For example the number 165 can be entered in a source file in the following different ways:

HEX	—	.a5	—	prefix = "."
BINARY	—	.n10100101	—	prefix = ".n"
DECIMAL	—	.m165	—	prefix = ".m"
OCTAL	—	.o245	—	prefix = ".o"

2) ASSEMBLER DIRECTIVE : SET ADDRESS POINTER — The user must set the assembler address pointer in the first command line of the source file so that the assembler will know where to start the assembly. This is done in the following manner:

```
10 REM [.m53000
```

NOTE that the open square bracket means SET THE ASSEMBLER ADDRESS POINTER TO THE FOLLOWING ADDRESS and that address can be written in any valid number system.

3) SOURCE LINE FORMAT — Each line of the source file must have a line number, the basic word REM, a space followed by an assembler directive or a Z80 MNEMONIC with appropriate addresses and numbers entered

eg. 200 REM ld a,.10

Multiple statements may be entered in the same line but the statements must be separated by a single quote.

eg. 150 REM ld a,.m15'ld b,.0a'add a,b

4) ASSEMBLER DIRECTIVE : COMMENT — Comments may be entered in any line of your source file following a ! sign. The assembler ignores any text after the ! sign and moves on to the next source line.

eg. 100 REM ! subroutine to print string
110 REM ld a,.m65'! character code into register A

5) ASSEMBLER DIRECTIVE : LABELS — Labels may be placed at any point in your source file and the label will be equivalent to the assembler address pointer at that point. Such labels may be used to address JUMPS, CALLS, LOADS or any other Z80 commands which require an address.

eg. 10 REM [.d000'Start

NOTE that labels can be a maximum of 5 characters long and the first character must be a capital letter with the remaining characters in lower case.

6) ASSEMBLER DIRECTIVE : NUMBER STORAGE — You can set aside storage areas for numeric constants and variables by using the directives db (single byte) or dw (two bytes).

eg. 200 REM Store'db .0a
210 REM Stor2'dw .m1000

7) ASSEMBLER DIRECTIVE : STRING STORAGE — You can set aside a storage area for strings by using the $ prefix.

eg. 150 REM Str1'$"This is a string":

8) ASSEMBLER DIRECTIVE : END MARKER — The close square bracket is used to mark the end of the source file.

eg. 1000 REM]

9) SETTING JUMP ADDRESSES — Many assemblers use the EQU statement to set up labels with external addresses (ie. addresses outside the current MC program — eg. ROM routines.). With the SUPER ASSEMBLER you use the open square bracket to set the address pointer and then specify the label. This must be done at the start of the source file.

eg. 10 REM [.394d'Chput
20 REM [.403d'Chget

10) TO RUN THE ASSEMBLER — Having entered the source file type in the following:

DEFUSR 0 = 51830 followed by ENTER

now type Z = USR 0 (0) to start assembly.

To RUN your MC program DEFUSR 1 = YOUR PROGRAM START ADDRESS and then Z = USR 1 (0) to execute your program.

11) SAVING YOUR MC PROGRAMS — Use the standard BSAVE command to save the machine code and use SAVE or CSAVE to save the source files.

eg. BSAVE "mcprog" ,start address,end address,run address
CSAVE "source"

12) FINAL CAUTIONS:

a) There MUST be a space between the REM and the instruction.

b) All numbers (except the line numbers) MUST be properly prefixed.

c) All instructions MUST be in lower case.

d) All labels MUST have the first letter in upper case and the remaining letters in lower case.

e) The first instruction MUST be the open square bracket.

f) The last instruction MUST be the close square bracket.

g) BEWARE of overwriting the machine area at the top of memory.

h) BEWARE of overwriting the SUPER ASSEMBLER.

i) Always reserve space before loading your MC programs.

j) Always SAVE your source files before assembling and running — the mc program may crash and you will have to retype the source from the start.

****

Sample source files can be found in the next few chapters. Work through each of the files and exercises to gain an appreciation of machine code in general and the SUPER ASSEMBLER operation in particular.

Most of the source files make use of BASIC ROM ROUTINES. Each time a new routine is used the function of the routine is highlighted as in the following example:

Chput		
USE	=	print character to screen
ADDRESS	=	00a2 hex
ENTRY	=	character code in A
EXIT	=	none
CHANGES	=	no registers changed

CHAPTER 18

SIMPLE SCREEN ROUTINES

Load up the assembler and then set up as follows:

type DEF USR 0 = 51830
type DEF USR 1 = &HD000

Now type in the following source file or load it from tape:

SOURCE FILE 18.1

```
10 REM ! Chput demo
20 REM !
30 REM !
40 REM [.00a2'Chput'!        print character rom routine
50 REM [.d000'!              assembly start address
60 REM ld a,.m65'!           code for A into the A register
70 REM call Chput'!          print it
80 REM ret'!                 return to basic
90 REM ]'!                   end of source
```

Type:

Z = USR 0(0) to assemble the file at address &HD000.
Z = USR 1(0) to run the MC program.

When you run the program you will notice that it prints A on the screen.

Notice that the ASCII code of the character to be printed must be in the A register before calling the CHPUT routine.

Chput		
USE	=	print character to screen
ADDRESS	=	00a2 hex
ENTRY	=	character code in A
EXIT	=	none
CHANGES	=	no registers changed

SOURCE FILE 18.2 is a modified version of SOURCE FILE 18.1 — the program continuously loops and fills the screen with characters. Notice the label Loop in line 80 — this label marks the relative jump destination for the jump in line 110.

SOURCE FILE 18.2

```
 10 REM ! perpetual demo
 20 REM !
 30 REM !
 40 REM [.00a2'Chput'!          print character rom routine
 50 REM [.d000'!                assembly start address
 60 REM ld a,.m65'!             code for A into the A
 70 REM !                               register
 80 REM Loop '!                 Loop start point
 90 REM !
100 REM call Chput'!            print character in A register
110 REM jr Loop'!               unconditional relative jump to Loop
120 REM ret'!                   return to basic
130 REM ]'!                     end of source
```

Now assemble [Z = USR 0(0)] and run [Z = USR 1(0)]and notice how quickly the screen fills up with the letter A.

You have probably noticed also that this program continues to run on and on and on ....

In fact this program will run on for ever or until you switch your computer off. We did not provide for the program to reach an end either automatically or by a CTRL/STOP.

Switch the computer off and then on again — load up the assembler and then load SOURCE FILE 18.2. We will now modify the file to allow CTRL/STOP to be used.

Source file 18.3 is a modified version of file 18.2 but at each Loop the computer checks if CTRL/STOP has been pressed and ends the program if the check is positive.

SOURCE FILE 18.3

```
 10 REM ! CTRL/STOP demo
 20 REM !
 30 REM !
```

SOURCE FILE 18.3 CONTINUED

```
 40 REM [.00a2'Chput'!        print character rom routine
 50 REM [.00b7'Break'!        check CTRL/STOP rom routine
 60 REM [.d000'!              assembly start address
 70 REM !
 80 REM Loop '!               Loop start point
 90 REM !
100 REM ld a,.m65'!           code for A into the A register
110 REM call Chput'!          print it
120 REM call Break'!          check for CTRL/STOP
130 REM jr nc,Loop'!          no CTRL/STOP so Loop
140 REM ret'!                 CTRL/STOP so return to basic
150 REM ]'!                   end of source
```

Notice that the relative jump to Loop has changed to a conditional relative jump. The routine BREAK sets the carry flag if CTRL/STOP has been pressed so we jump to LOOP only if the carry flag is not set (jump relative non carry).

Save the source file, assemble it, and then run the mc program. Assembly and execution of this (and all other source files in this book) is performed in the same manner.

Break		
USE	=	check for CTRL/STOP
ADDRESS	=	00b7 hex
ENTRY	=	none
EXIT	=	carry flag set if CTRL/STOP
CHANGES	=	A and F registers

Getting a little tired of a screen full of A's? — then try the next program — SOURCE FILE 18.4.

SOURCE FILE 18.4

```
10 REM ! Chget demo
20 REM !
30 REM !
40 REM [.00a2'Chput'!         print character rom routine
50 REM [.00b7'Break'!         check CTRL/STOP rom routine
```

```
 60 REM [.009f'Chget'!        get character from keyboard
 70 REM [.d000'!              assembly start address
 80 REM call Chget'!          get character in A register
 90 REM push af'!             save A register on stack
100 REM !
110 REM Loop '!               Loop start point
120 REM !
130 REM pop af'!              recover A register from stack
140 REM call Chput'!          print character
150 REM push af'!             save A register on stack
160 REM call Break'!          check for CTRL/STOP
170 REM jr nc,Loop'!          no CTRL/STOP so Loop
180 REM pop af'!              CTRL/STOP so clear A off stack
190 REM ret'!                 return to basic
200 REM ]'!                   end of source
```

This time a new ROM ROUTINE called Chget is used. When you assemble and run the program nothing will happen until you press a key. The routine Chget waits until a key is pressed and our mc program will then print a screen full of your selected character.

Chget		
USE	=	get character from keyboard
ADDRESS	=	009f hex
ENTRY	=	none
EXIT	=	character in the A register
CHANGES	=	A and F registers

The final source file in this chapter creates a different type of display which is sometimes known as a BARBER POLE display. This display prints the character set over and over again by incrementing the character code at each Loop.

SOURCE FILE 18.5

```
10 REM ! Chsns demo
20 REM !
30 REM !
40 REM [.00a2'Chput'!         print character rom routine
```

SOURCE FILE 18.5 CONTINUED

```
 50 REM [.009c'Chsns'!      check any key rom routine
 60 REM [.d000'!            assembly start address
 70 REM !
 80 REM Start'!             Start routine address
 90 REM !
100 REM ld a,.m31'!         space code — 1 into A
                            register
110 REM push af'!           save A register on stack
120 REM !
130 REM Loop'!              Loop routine address
140 REM !
150 REM pop af'!            recover A register from stack
160 REM inc a'!             increase character code
170 REM cp .m126'!          last ascii character y/n ?
180 REM jr z, Start'!       yes so back to Start
190 REM call Chput'!        no so print character
200 REM push af'!           save A register on stack
210 REM call Chsns'!        key press y/n?
220 REM jr z,Loop'!         no so Loop
230 REM pop af'!            yes so clear A off stack
240 REM ret'!               return to basic
250 REM ]'!                 end of source
```

To stop the display simply press any key — this feature is supplied by the ROM ROUTINE Chsns which checks for a key press at each Loop. The ZERO FLAG is set if there has NOT been a key press.

Chsns		
USE	=	check for a key press
ADDRESS	=	009c hex
ENTRY	=	none
EXIT	=	zero flag set if no key press
CHANGES	=	A and F registers

CHAPTER 19

MORE PRINTING ROUTINES

The source file 19.1 uses the basic PRINT routine to print a string on to the screen.

SOURCE FILE 19.1

```
10 REM ! Print routine demo
20 REM !
30 REM !
40 REM [.4a24'Print'!              Print routine address
50 REM [.d000'!                    assembly start address
60 REM ld hl,Str'!                 set HL register to point to Str
70 REM call Print'!                Print Str
80 REM ret'!                       return to basic
90 REM Str'$"This is a string":']'! set up string Str
```

Note that the string is printed at the current cursor position and all other text remains on the screen. The SYNTAX of the string in line 90 is very important — a string must be enclosed in double quotes and must end with a colon or a zero byte. The $ sign at the start of the string is the assembler directive to indicate that a string follows.

When calling the routine PRINT from basic you should use the syntax Z$ = USR1(0) and not the usual Z = USR1(0).

Print		
USE	=	print a string to the screen
ADDRESS	=	4a24 hex
ENTRY	=	HL points to string address
EXIT	=	none

SOURCE FILE 19.2

```
10 REM ! RST 18 hex demo
20 REM !
30 REM !
40 REM [.4a24'Print'!              Print routine address
```

SOURCE FILE 19.2 CONTINUED

```
 50 REM [.d000'!                    assembly start address
 60 REM ld a,.0c'!                  clear screen character into A
 70 REM rst .18'!                   put character in A to screen
 80 REM ld hl,Str'!                 set HL register to point to Str
 90 REM call Print'!                Print Str
100 REM ret'!                       return to basic
110 REM Str'$"This is a string":]'! set up string Str
```

Source file 19.2 illustrates one of the useful restart instructions of the MSX computers — rst 18 is a single byte instruction which is used to print the character in the A register onto the screen. In file 19.2 the character printed is the clear screen character CHR$(0C) but you can put any character into the A register and print it with a rst 18. Note that the 18 in the rst 18 is in HEX and not decimal.

Rst 18 behaves in the same manner as the rom routine Chput with no registers affected by the instruction.

Other useful characters to print for screen formatting are:

CHR$(09)	=	TAB CURSOR
CHR$(0A)	=	LINE FEED
CHR$(0C)	=	CLEAR SCREEN
CHR$(0D)	=	CARRIAGE RETURN
CHR$(1C)	=	CURSOR 1 SPACE TO THE RIGHT
CHR$(1D)	=	CURSOR 1 SPACE TO THE LEFT
CHR$(1E)	=	CURSOR 1 LINE UP
CHR$(1F)	=	CURSOR 1 LINE DOWN

The final source file in this chapter is file 19.3 — in this file you will see how to position the cursor at any point on the screen before printing your text. The technique uses the rom routine "Posit" with the cursor X (across) position in the H register and the cursor Y position in the L register.

SOURCE FILE 19.3

```
 10 REM ! cursor position demo
 20 REM !
 30 REM !
 40 REM [.00c6'Posit'!          cursor position routine
 50 REM [.4a24'Print'!          print string
 60 REM !
 70 REM [.d000'!                assembly address
 80 REM !
 90 REM !
100 REM ld a,.m12'!             clear screen character into A
110 REM rst .18'!               clear screen
120 REM ld h,.m12'!             cursor column (across)
130 REM ld l,.m10'!             cursor row (down)
140 REM call Posit'!            position cursor
150 REM ld hl,Mesg'!            set HL to point to message
160 REM call Print'!            print message
170 REM ret'!                   return to basic
180 REM Mesg'!                  message address label
190 REM $"test message." '!     message string
200 REM db .0'!                 message end marker
210 REM ]'!                     end of source
```

Posit		
USE	=	locate cursor on the screen
ADDRESS	=	00c6 hex
ENTRY	=	X position in H Y position in L
EXIT	=	none
CHANGES	=	A and F registers changed

CHAPTER 20

THE SOUND OF MUSIC

The first source file in this section shows how to use the basic command PLAY from machine code. The HL register pair is used to point to the location of the music string and then the play routine is called.

Assemble the file in the normal way and then DEF USR1 = &HD000. To PLAY the music type Z = USR1(0) followed by ENTER.

SOURCE FILE 20.1

```
10 REM ! Play routine demo
20 REM !
30 REM !
40 REM [.73e5'Play'!              Play routine address
50 REM [.d000'!                   assembly start address
60 REM ld hl,Str'!                set HL register to point to Str
70 REM call Play'!                Play Str
80 REM ret'!                      return to basic
90 REM Str'$"abcabccbd":']'!      set up music string Str
```

NOTE that the music string must be written in the same way as a text string ie. enclosed in double quotes and terminated with a colon or a zero byte.

Play		
USE	=	play a music string
ADDRESS	=	73e5hex
ENTRY	=	HL points to string address
EXIT	=	none

One of the more interesting features of the sound chip is the repeat facility — To make a sound repeat continually you must set bit 3 of register 13. When such a sound is initialised it will continue repeating until interrupted by a CTRL/STOP or another SOUND command. The repeating sound is controlled completely by the sound chip — the computer may continue with other activities without disturbing the SOUND.

Source file 20.2 shows how to intitialise a repeating sound from machine code.

SOURCE FILE 20.2

```
 10 REM ! sound demo          (steam train)
 20 REM !
 30 REM !
 40 REM [.96'Rdpsg'!          read from PSG
 50 REM [.93'Wtpsg'!          write to PSG
 60 REM [.d000'!              assembly start address
 70 REM ld a,.m7'!            PSG register 7 into A
 80 REM call Rdpsg'!          read current value (reg 7)
 90 REM and .n11000000'!      extract bits 6 and 7
100 REM add .n00110111'!      add "on switch" noise channel A
110 REM ld e,a'!              data into E register
120 REM ld a,.m7'!            PSG register 7 into A
130 REM call Wtpsg'!          write data to PSG
140 REM ld a,.m8'!            PSG register 8 into A
150 REM ld e,.n00011111'!     volume for noise channel A
160 REM call Wtpsg'!          write to PSG
170 REM ld a,.m12'!           PSG register 12 into A
180 REM ld e,.n00000011'!     envelope period (coarse)
190 REM call Wtpsg'!          write to PSG
200 REM ld a,.m13'!           PSG register 13 into A
210 REM ld e,.n00001110'!     envelope shape
220 REM call Wtpsg'!          write to PSG
230 REM ret'!                 return to basic
240 REM ]'!                   end of source
```

The sound chip register 7 deserves a special mention — in this register the 6 lower bits are used to enable the sound and tone channels whilst the upper two bits are used in conjunction with the sound chip ports A and B. It is therefore desirable to preserve these two bits when sound channels are enabled. Lines 70 — 90 in source file 20.2 preserve the two upper bits and enable the noise channel A of the sound chip.

Rdpsg		
USE	=	to read data from PSG
ADDRESS	=	0096 hex
ENTRY	=	A contains PSG register number
EXIT	=	A contains data

Wtpsg		
USE	=	to write data to PSG
ADDRESS	=	0093 hex
ENTRY	=	A contains PSG register number
EXIT	=	none

****

PROGRAM LIST 20.3

```
 10 REM sound demo          (space ship)
 20 REM
 30 REM
 40 SOUND0,&B11100000      'tone period channel A (fine)
 50 SOUND2,&B11111111      'tone period channel B (fine)
 60 SOUND7,&B00111100      'enable tone channels A and B
 70 SOUND8,&B00011111      'volume channel A
 80 SOUND9,&B00000111      'volume channel B
 90 SOUND12,&B00000011     'envelope period
100 SOUND13,&B00001100     'envelope shape
```

Program file 20.3 is another example of repeating sound — this time using two tone channels. The program is written in basic but you can convert it to machine code as an exercise (REMEMBER the rules for register 7).

****

CHAPTER 21

TRANSFERING VARIABLES FROM MACHINE CODE TO BASIC

Machine code routines are often used to speed up certain operations which would take a long time in basic. When MC is used in this way it.is usually necessary to transfer some results back to the basic program.

Such results can be placed into known memory locations and then PEEKED by the basic program. A more elegant way of returning results is for the MC program to place the result directly into a basic variable. Source file 21.1 illustrates this method of returning results.

The following two new ROM ROUTINES are used:

Eval		
USE	=	evaluate a basic expression
ADDRESS	=	4c64 hex
ENTRY	=	HL points to expression
EXIT	=	result type in Vtyp result in Dac

Vtyp is the system variable which contains the type of result returned by the expression evaluator:

Vtyp address = f663 hex
Vtyp contents = 2 for integer result.
= 3 for string result.
= 4 for single precision result.
= 8 for double precision result.

Fac is the floating point accumulator which contains the result returned by the expression evaluator:

Fac address = f7f6 hex
Fac contents — integer result contained in Fac + 2 and Fac + 3

Fac contents — with a string result the address of the 3 byte string descriptor is contained in Fac + 2 and Fac + 3.
— single precision result is in Fac to Fac + 3.
— double precision result is in Fac to Fac + 7.

* *

Vget		
USE	=	get address of variable
ADDRESS	=	5ea4 hex
ENTRY	=	HL points to variable name
EXIT	=	DE points to variable address
CHANGES	=	B and C registers

NOTE that if the variable does not exist then Vget will create it. Default precision will be used for the variable unless precision is stated as part of the variable name (eg. A#or B!). In source file 21.1 the variable used is AD and the variable is forced to the correct precision by updating the variable definition table.

SOURCE FILE 21.1

```
 10 REM ! returning variables to basic
 20 REM !
 30 REM !
 40 REM [.4c64'Eval'!        expression evaluator
 50 REM [.5ea4'Vget'!        get address of variable
 60 REM [.f663'Vtyp'!        system variable value type
 70 REM [.f7f6'Fac'!         floating point acc.
 80 REM [.f6ca'Vdef'!        variable definition table
 90 REM !
100 REM !
110 REM [.d000'!             assembly start address
120 REM ld hl,Exp'!          point HL to expression
130 REM call Eval'!          evaluate it
140 REM ld a,(Vtyp)'!        value type into A register
150 REM ld (Vdef),a'!        force variable to valtype
```

160 REM ld b,.0'ld c,a'!	variable length into BC
170 REM ld hl,Varn'!	point HL to variable name
180 REM push bc'!	save BC on the stack
190 REM call Vget'!	get variable position
200 REM pop bc'!	recover BC from the stack
210 REM ld hl,Fac'!	point HL to Fac
220 REM ld a,.2'cp c'!	is the value an integer?
230 REM jr nz,Stor'!	no so goto Stor
240 REM inc hl'inc hl'!	yes so increase Fac pointer by two
250 REM !	
260 REM !	
270 REM Stor'!	subroutine to move value to variable
280 REM !	
290 REM ldir'!	move it
300 REM ret'!	return to basic
310 REM Exp'!	expression Exp
320 REM db .ff'db .m148'!	basic tokens for VAL
330 REM $("132435.8866")'!	string for VAL to operate on
340 REM db .0'!	expression end marker
350 REM !	
360 REM !	
370 REM Varn'!	variable name label
380 REM !	
390 REM $AD:'!	variable name = AD
400 REM !	
410 REM]'!	end of source file

****

CHAPTER 22

SOME GRAPHICS ROUTINES

One of the questions I am asked most often is — HOW DO YOU SCROLL THE GRAPHICS SCREEN?

There is obviously no quick and simple answer to this question and so my usual reply is — WITH DIFFICULTY — and then I go on to explain as follows:

1) Move the graphics name table from the video ram into the normal ram.
2) Rotate the lines of the name table one byte to the right or left.
3) Move the adjusted name table from the normal ram back to its normal position in the video ram.
4) The procedure in basic is much too slow and so machine code must be used in order to get a smooth scrolling effect.

This procedure is illustrated in source files 22.1 and 22.2. Notice that the files have a machine code source section and a pure basic section. Assemble the files in the normal way and then type RUN followed by ENTER — the basic section will first draw a simple picture on the screen and then repeatedly call the mc program to scroll the top two thirds of the screen.

Ldvm		
USE	=	move a block of data from memory to VRAM
ADDRESS	=	005c hex
ENTRY	=	Address of source in HL Address of destination in DE Number of bytes in BC
EXIT	=	none
CHANGES	=	all registers

Ldmv		
USE	=	move a block of data from VRAM to memory
ADDRESS	=	0059 hex
ENTRY	=	Address of source in HL Address of destination in DE Number of bytes in BC
EXIT	=	none
CHANGES	=	all registers

Source file 22.1 scrolls the screen to the left.

SOURCE FILE 22.1

```
 10 REM !                   screen 2 left scroll
 20 REM !
 30 REM !
 40 REM !                   collect name table
 50 REM !
 60 REM [.5c'Ldvm'!         move memory to VRAM
 70 REM [.59'Ldmv'!         move VRAM to memory
 80 REM [.d000'!            assembly start address
 90 REM ld hl,.1800'!       VRAM source
100 REM ld de,.d100'!       memory destination
110 REM ld bc,.200'!        number of bytes
120 REM call Ldmv'!         fetch data
130 REM !
140 REM Scrol'!             subroutine to scroll the name table
150 REM !
160 REM ld de,.d100'!       buffer start
170 REM ld hl,.d101'!       start + 1
180 REM ld bc,.001f'!       line length - 1
190 REM Loop
200 REM push bc'!           save BC on stack
210 REM ld a,(de)'!         first byte into A
220 REM ldir'!              move line one to left
230 REM ld (de),a'!         first byte into last position
240 REM inc de'!            start of next line
250 REM inc hl'!            line start + 1
260 REM pop bc'!            recover line length
270 REM ld a,.d4'!          end check
280 REM cp h'!              is it the end?
290 REM jr nz,Loop'!        no so do it again
300 REM !
310 REM Send'!              yes so send to VRAM
320 REM !
330 REM ld hl,.d100'!       memory source into HL
340 REM ld de,.1800'!       VRAM destination into DE
350 REM ld bc,.200'!        byte count
360 REM call Ldvm'!         transfer to VRAM
370 REM ret'!               return to basic
380 REM ]'!                 end of source
390 REM !
400 REM !
410 REM !
```

SOURCE FILE 22.1 CONTINUED

```
420 REM                   basic supt routine
430 REM !
440 REM !
450 REM !
460 ONSTOPGOSUB620:STOPON
470 COLOR15,8,8
480 SCREEN 2
490 OPEN"grp:"FOROUTPUTAS#1
500 PSET(0,80),1
510 DRAW"e90f45e29f80e30"
520 PAINT(4,80),1
530 LINE(10,140)—(150,190),11,BF
540 PSET(38,160),11
550 COLOR4
560 PRINT#1,"LEFT SCROLL"
570 DEFUSR2=&HD000
580 DEFUSR3=&HD00C
590 Y=USR2(0)
600 Y=USR3(0)
610 GOTO600
620 COLOR15,4,4
630 END
```

Source file 22.2 scrolls the screen to the right.

SOURCE FILE 22.2

```
 10 REM ! screen 2 right scroll
 20 REM !
 30 REM !
 40 REM ! collect name table
 50 REM !
 60 REM [.5c'Ldvm
 70 REM [.59'Ldmv
 80 REM [.d000
 90 REM ld hl,.1800
100 REM ld de,.d100
110 REM ld bc,.200
120 REM call Ldmv
130 !
140 REM Scroll
150 REM !
160 REM ld de,.d2ff
```

SOURCE FILE 22.2 CONTINUED

```
170 REM ld hl,.d2fe
180 REM ld bc,.001f
190 REM Loop
200 REM push bc
210 REM ld a,(de)
220 REM lddr
230 REM ld (de),a
240 REM dec de
250 REM dec hl
260 REM pop bc
270 REM ld a,.d0
280 REM cp h
290 REM jr nz,Loop
300 REM !
310 REM Send
320 REM !
330 REM ld hl,.d100
340 REM ld de,.1800
350 REM ld bc,.200
360 REM call Ldvm
370 REM ret
380 REM ]
390 REM !
400 REM !
410 REM !
420 REM      basic support routine
430 REM !
440 REM !
450 REM !
460 ONSTOPGOSUB620:STOPON
470 COLOR15,8,8
480 SCREEN2
490 OPEN"grp:"FOROUTPUTAS#1
500 PSET(0,80),1
510 DRAW"e90f45e29f80e30"
520 PAINT(4,80),1
530 LINE(10,140)-(150,190),11,BF
540 PSET(35,160),11
550 COLOR4
560 PRINT#1,"RIGHT SCROLL"
570 DEFUSR2=&HD000
580 DEFUSR3=&HD00C
590 Y=USR2(0)
600 Y=USR3(0)
610 GOTO600
620 COLOR15,4,4
630 END
```

Source file 22.3 is a machine code version of the sprite detection routine which was presented earlier in Basic.

The routine is self explanatory if read in conjunction with the earlier basic version — the sprite which caused the interrupt is returned in the basic variable A.

Source file 22.3 is executed in the same way as file 22.2.

SOURCE FILE 22.3

```
 20 REM ! sprite collision routine
 30 REM !
 40 REM !
 50 REM !
 60 REM [.4a'Rdvrm
 70 REM [.4d'Wtvrm
 80 REM [.87'Calat
 90 REM [.d000
100 REM ld b,.m31
110 REM Next
120 REM ld a,b
130 REM call Calat
140 REM push hl
150 REM call Rdvrm
160 REM pop hl
170 REM cp .m209
180 REM jr nz,Test
190 REM Dnext'djnz,Next
200 REM ret
210 REM !
220 REM Test
230 REM !
240 REM push bc
250 REM push af
260 REM ld a,.m209
270 REM push hl
280 REM call Wtvrm
290 REM halt
300 REM ld a,(.f3e7)
310 REM and .n00100000
320 REM pop hl
```

```
330 REM jr z,Found
340 REM pop af
350 REM push af
360 REM call Wtvrm
370 REM pop af
380 REM pop bc
390 REM jr Dnext
400 REM !
410 REM Found
420 REM !
430 REM pop af
440 REM pop bc
450 REM ld hl,Varn
460 REM push bc
470 REM call .5ea4
480 REM pop bc
490 REM ex de,hl
500 REM ld (hl),b
510 REM inc hl
520 REM ld (hl),.0
530 REM ret
540 REM Varn'$A'db .0']
550 REM
560 REM
570 REM basic support program
580 REM
590 REM
600 DEFINTA-Z
610 ONSTOPGOSUB820:STOPON
620 SCREEN1
630 DEFUSR2=&HD000
640 ONSPRITEGOSUB780
650 FORX=0TO7
660 A$=A$+CHR$(255)
670 NEXT
680 SPRITE$(0)=A$
690 FORP=1TO15
700 PUTSPRITEP,(50+P*10,P*10),P,0
710 NEXT
720 PD=(INT((RND(-TIME)*180)/10))*10
730 SPRITEON
740 FORZ=-20TO255
```

SOURCE FILE 22.3 CONTINUED

```
750 PUTSPRITE0,(Z,PD),3,0
760 NEXT
770 GOTO720
780 SPRITEOFF
790 Z=USR2(0)
800 PRINTA
810 RETURN720
820 SCREEN0
830 END
```

Calat		
USE	=	Find the address of a sprite attribute entry
ADDRESS	=	0087 hex
ENTRY	=	Sprite plane number in A
EXIT	=	Attribute address in HL
CHANGES	=	AF, DE and HL

****

This book was designed to provide the reader with an introduction to machine code on the MSX — interested readers can now build on this grounding using one of the many good Z80 books which are available in your local book store.

APPENDIX 1

Z80 MACHINE CODE MNEMONICS

In the next few pages you will find a full list of the Z80 mnemonics which you will use in machine code source files. In the list the following shorthand is used:

1) DIS means an 8 bit displacement which can range from 127 to minus 128.

2) NN means an 8 bit number which can range from 0 to 255.

3) HHLL means a 16 bit number which can range from 0 to 65535 — LL HH is the same number with the high and low bytes reversed.

4) ADDR means a memory address or label — DR AD is the address with the high and low bytes reversed as required by the Z80.

5) PORT means an input or output port with a number in the range 0 to 255.

6) All mnemonic instructions and register names are in lower case as required by the assembler. The object code is given in upper case hex numbers.

   REMEMBER that you type the source code into a source file and the assembler creates the object code.

****

Add with carry (8 bit)

The content of the carry flag (1 or 0) is added to the value in the "a" register and then the second named value (stated value or register contents or memory location contents) is added to the result. The final result is placed into the "a" register.

SOURCE CODE	OBJECT CODE
adc a,(hl)	8E
adc a,(ix + DIS)	DD 8E DIS
adc a,(iy + DIS)	FD 8E DIS
adc a,a	8F
adc a,b	88
adc a,c	89

SOURCE CODE	OBJECT CODE
adc a,d	8A
adc a,NN	CE NN
adc a,e	8B
adc a,h	8C
adc a,l	8D

****

Add with carry (16 bit)

The content of the carry flag (1 or 0) is added to the contents of the "hl" register and then the second named value (register pair contents) is added to the result. The final result is placed into the "hl" register.

SOURCE CODE	OBJECT CODE
adc hl,bc	ED 4A
adc hl,de	ED 5A
adc hl,hl	ED 6A
adc hl,sp	ED 7A

Add instructions (8 bit)

The second named value (stated value or register contents or memory location contents) is added to the value in the "a" register and the result is placed into the "a" register.

SOURCE CODE	OBJECT CODE
add a,(hl)	86
add a,(ix+DIS)	DD 86 DIS
add a,(iy+DIS)	FD 86 DIS
add a,a	87
add a,b	80
add a,c	81
add a,d	82
add a,NN	C6 NN
add a,e	83
add a,h	84
add a,l	85

****

Add instruction (16 bit)

The contents of the second named register pair are added to the contents of the first named register pair. The result is placed into the first named register pair.

SOURCE CODE	OBJECT CODE
add hl,bc	09
add hl,de	19
add hl,hl	29
add hl,sp	39
add ix,bc	DD 09
add ix,de	DD 19
add ix,ix	DD 29
add ix,sp	DD 39
add iy,bc	FD 09
add iy,de	FD 19
add iy,iy	FD 29
add iy,sp	FD 39

****

Logical "and" instructions

A logical "and" operation is performed between the named value (specified value, register contents or memory location contents) and the contents of the "a" register. The result is placed into the "a" register.

SOURCE CODE	OBJECT CODE
and (hl)	A6
and (ix+DIS)	DD A6 DIS
and (iy+DIS)	FD A6 DIS
and a	A7
and b	A0
and c	A1
and d	A2
and NN	E6 NN
and e	A3
and h	A4
and l	A5

****

Logical "and" is a bit by bit comparison between two 8 bit numbers. If a particular bit is 1 in both numbers then the corresponding bit in the result will also be one otherwise the result bit will be zero.

These instructions are useful for extracting selected parts of numbers — eg. 01010101 and 00001111 = 00000101 — the lower 4 bits of the first number are extracted by masking off the upper 4 bits.

****

Bit testing instructions

These instructions test the condition of a specified bit in a specified memory location or register. The zero flag is set according to the result of the test and so a zero conditional instruction usually follows the bit test instruction.

SOURCE CODE	OBJECT CODE
bit .0,(hl)	CB 46
bit .0,(ix+DIS)	DD CB NN 46
bit .0,(iy+DIS)	FD CB NN 46
bit .0,a	CB 47
bit .0,b	CB 40
bit .0,c	CB 41
bit .0,d	CB 42
bit .0,e	CB 43
bit .0,h	CB 44
bit .0,l	CB 45
bit .1,(hl)	CB 4E
bit .1,(ix+DIS)	DD CB NN 4E
bit .1,(iy+DIS)	FD CB NN 4E
bit .1,a	CB 4F
bit .1,b	CB 48
bit .1,c	CB 49
bit .1,d	CB 4A
bit .1,e	CB 4B
bit .1,h	CB 4C
bit .1,l	CB 4D

SOURCE CODE	OBJECT CODE
bit .2,(hl)	CB 56
bit .2,(ix+DIS)	DD CB NN 56
bit .2,(iy+DIS)	FD CB NN 56
bit .2,a	CB 57
bit .2,b	CB 50
bit .2,c	CB 51
bit .2,d	CB 52
bit .2,e	CB 53
bit .2,h	CB 54
bit .2,l	CB 55
bit .3,(hl)	CB 5E
bit .3,(ix+DIS)	DD CB NN 5E
bit .3,(iy+DIS)	FD CB NN 5E
bit .3,a	CB 5F
bit .3,b	CB 58
bit .3,c	CB 59
bit .3,d	CB 5A
bit .3,e	CB 5B
bit .3,h	CB 5C
bit .3,l	CB 5D
bit .4,(hl)	CB 66
bit .4,(ix+DIS)	DD CB NN 66
bit .4,(iy+DIS)	FD CB NN 66
bit .4,a	CB 67
bit .4,b	CB 60
bit .4,c	CB 61
bit .4,d	CB 62
bit .4,e	CB 63
bit .4,h	CB 64
bit .4,l	CB 65
bit .5,(hl)	CB 6E
bit .5,(ix+DIS)	DD CB NN 6E
bit .5,(iy+DIS)	FD CB NN 6E
bit .5,a	CB 6F
bit .5,b	CB 68
bit .5,c	CB 69
bit .5,d	CB 6A
bit .5,e	CB 6B
bit .5,h	CB 6C
bit .5,l	CB 6D

SOURCE CODE	OBJECT CODE
bit .6,(hl)	CB 76
bit .6,(ix+DIS)	DD CB NN 76
bit .6,(iy+DIS)	FD CB NN 76
bit .6,a	CB 77
bit .6,b	CB 70
bit .6,c	CB 71
bit .6,d	CB 72
bit .6,e	CB 73
bit .6,h	CB 74
bit .6,l	CB 75
bit .7,(hl)	CB 7E
bit .7,(ix+DIS)	DD CB NN 7E
bit .7,(iy+DIS)	FD CB NN 7E
bit .7,a	CB 7F
bit .7,b	CB 78
bit .7,c	CB 79
bit .7,d	CB 7A
bit .7,e	CB 7B
bit .7,h	CB 7C
bit .7,l	CB 7D

****

Call instructions

Call instructions work like a basic GOSUB — a return address is automatically pushed onto the stack and the program counter is set to the call address. At the end of the called subroutine a return instruction pops the return address off the stack and into the program counter.

SOURCE CODE	OBJECT CODE
call ADDR	CD DR AD — unconditional
call c,ADDR	DC DR AD — if carry flag set
call m,ADDR	FC DR AD — if sign flag is set
call nc,ADDR	D4 DR AD — if carry flag is reset
call nz,ADDR	C4 DR AD — if zero flag is reset
call p,ADDR	F4 DR AD — if sign flag is reset
call pe,ADDR	EC DR AD — if parity flag is set
call po,ADDR	E4 DR AD — if parity flag is reset
call z,ADDR	CC DR AD — if the zero flag is set

****

Compare instructions

A value or the contents of the specified register or memory location are compared to the contents of the "a" register and the CPU flags are set as if a subtraction from the "a" register had occurred. Testing the flags after a compare instruction provides information concerning the compared value.

SOURCE CODE	OBJECT CODE
cp (hl)	BE
cp (ix + DIS)	DD BE DIS
cp (iy + DIS)	FD BE DIS
cp a	BF
cp b	B8
cp c	B9
cp d	BA
cp NN	FE NN
cp e	BB
cp h	BC
cp l	BD

****

Special block search instructions

The "hl" register pair is set up to point to the first byte in the search area. The register pair "bc" contains the number of bytes in the search area. The "a" register contains the value which is to be found in the search area. The contents of the byte (pointed to by hl) is compared to the contents of the "a" register and the cpu flags are set accordingly. The "bc" register decrements and the "hl" register increments or decrements according to which instruction is used. With the repeat instructions the operations will repeat until the "bc" register contains zero or until an exact match is found between the byte indicated by "hl" and the contents of the "a" register.

SOURCE CODE	OBJECT CODE
cpd	ED A9 — decrement hl and bc
cpdr	ED B9 — decrement hl and bc then repeat
cpi	ED A1 — increment hl and decrement bc
cpir	ED B1 — as "cpi" but with repeat

The decrement instructions

The contents of a memory byte, 8 bit register, or 16 bit register are decreased by one.

SOURCE CODE	OBJECT CODE
dec (hl)	35
dec (ix+DIS)	DD 35 DIS
dec (iy+DIS)	FD 35 DIS
dec a	3D
dec b	05
dec bc	0B
dec c	0D
dec d	15
dec de	1B
dec e	1D
dec h	25
dec hl	2B
dec ix	DD 2B
dec iy	FD 2B
dec l	2D
dec sp	3B

****

The exchange instructions

Exchanges the contents of the indicated registers with the contents of the stack at the current stack pointer position. An instruction is also provided to exchange the contents of the "de" and "hl" registers.

SOURCE CODE	OBJECT CODE
ex (sp),hl	E3
ex (sp),ix	DD E3
ex (sp),iy	FD E3
ex de,hl	EB

****

Register bank exchanges

Two instructions are provided — one to exchange the "af" register banks and the other to exchange the "hl", "bc", and "de" banks.

SOURCE CODE	OBJECT CODE
ex af,af"	08 — exchange af
exx	D9 — exchange all but af

****

Input instructions

Input an 8 bit value through the specified input port into the specified register. Most of the instructions require that the input port number is in the "c" register.

SOURCE CODE	OBJECT CODE
in a,(c)	ED 78
in a,(PORT)	DB PORT
in b,(c)	ED 40
in c,(c)	ED 48
in d,(c)	ED 50
in e,(c)	ED 58
in h,(c)	ED 60
in l,(c)	ED 68

****

Block input

Values are input through the port specified in register "c" and placed into the memory byte pointed to by "hl". Register "b" is used as a counter and the value in "b" is decremented. The register pair "hl" is incremented or decremented depending on which instruction is used. With the repeat instructions the sequence of repeats will terminate when register "b" contains zero.

SOURCE CODE	OBJECT CODE
ind	ED AA — decrement hl
indr	ED BA — decrement hl and repeat
ini	ED A2 — increment hl
inir	ED B2 — increment hl and repeat

****

Increment instructions

The contents of the specified register or memory byte are increased by one.

SOURCE CODE	OBJECT CODE
inc (hl)	34
inc (ix+DIS)	DD 34 DIS
inc (iy+DIS)	FD 34 DIS
inc a	3C
inc b	04
inc bc	03
inc c	0C
inc d	14

SOURCE CODE	OBJECT CODE
inc de	13
inc e	1C
inc h	24
inc hl	23
inc ix	DD 23
inc iy	FD 23
inc l	2C
inc sp	33

****

Some unclassified instructions

SOURCE CODE	OBJECT CODE	ACTION
ccf	3F	flip the carry flag
scf	37	set carry flag to 1
cpl	2F	flip the bits in "a"
daa	27	decimal adjust "a"
di	F3	disable interrupts
ei	FB	enable interrupts
halt	76	stop operation until interrupt
im .0	ED 46	interrupt mode 0
im .1	ED 56	interrupt mode 1
im .2	ED 5E	interrupt mode 2
neg	ED 44	flip "a" then add 1
nop	00	no operation

****

Jump instructions

The program counter is set to the specified jump address if the flag condition (if any) is fulfilled.

SOURCE CODE	OBJECT CODE	
jp (hl)	E9	— unconditional
jp (ix)	DD E9	— unconditional
jp (iy)	FD E9	— unconditional
jp ADDR	C3 DR AD	— unconditional
jp c,ADDR	DA DR AD	— if carry flag is set
jp m,ADDR	FA DR AD	— if sign flag set
jp nc,ADDR	D2 DR AD	— if carry flag is reset
jp nz,ADDR	C2 DR AD	— if zero flag is reset
jp p,ADDR	F2 DR AD	— if sign flag is reset
jp pe,ADDR	EA DR AD	— if parity flag is set
jp po,ADDR	E2 DR AD	— if parity flag is reset
jp z,ADDR	CA DR AD	— if zero flag is set

NOTE that the parity flag checks the number of bits in "a" which are set to 1.

Parity odd(po) = odd number of bits.
Parity event(pe) = even number of bits.

Parity checks are often used to detect errors in data transfer operations.

****

Jump relative instructions

The displacement is added to the address in the program counter and the program counter is set to the new address if the flag conditions (if any) are fulfilled.

SOURCE CODE	OBJECT CODE	
jr c,DIS	38 DIS	— if carry flag is set
jr DIS	18 DIS	— unconditional
jr nc,DIS	30 DIS	— if carry flag is reset
jr nz,DIS	20 DIS	— if zero flag is reset
jr z,DIS	28 DIS	— if zero flag is set

NOTE that the SUPER ASSEMBLER accepts values or lables as displacements and addresses.

****

Instructions to load data into memory bytes

8 bit data is loaded from a register into the specified address. With 16 bit data the low byte is loaded into the specified address and the high byte is loaded into the address plus 1. Addressing is by direct (numeric address or label) or indirect by using a register pair as a pointer.

SOURCE CODE	OBJECT CODE	
ld (ADDR),a	32 DR AD	— 8 bit direct
ld (ADDR),bc	ED 43 DR AD	— 16 bit direct
ld (ADDR),de	ED 53 DR AD	— 16 bit direct
ld (ADDR),hl	ED 63 DR AD	— 16 bit direct
ld (ADDR),hl	22 DR AD	— 16 bit direct
ld (ADDR),ix	DD 22 DR AD	— 16 bit direct
ld (ADDR),iy	FD 22 DR AD	— 16 bit direct
ld (ADDR),sp	ED 73 DR AD	— 16 bit direct

The remaining instructions in this section are all 8 bit loads with indirect addressing.

SOURCE CODE	OBJECT CODE
ld (bc),a	02
ld (de),a	12
ld (hl),a	77
ld (hl),b	70
ld (hl),c	71
ld (hl),d	72
ld (hl),NN	36 NN
ld (hl),e	73
ld (hl),h	74
ld (hl),l	75
ld (ix+DIS),a	DD 77 DIS
ld (ix+DIS),b	DD 70 DIS
ld (ix+DIS),c	DD 71 DIS
ld (ix+DIS),d	DD 72 DIS
ld (ix+DIS),NN	DD 36 DIS NN
ld (ix+DIS),e	DD 73 DIS
ld (ix+DIS),h	DD 74 DIS
ld (ix+DIS),l	DD 75 DIS
ld (iy+DIS),a	FD 77 DIS
ld (iy+DIS),b	FD 70 DIS
ld (iy+DIS),c	FD 71 DIS
ld (iy+DIS),d	FD 72 DIS
ld (iy+DIS),NN	FD 36 DIS NN
ld (iy+DIS),e	FD 73 DIS
ld (iy+DIS),h	FD 74 DIS
ld (iy+DIS),l	FD 75 DIS

****

Register load instructions

Data is loaded from the source (value, memory byte, or register) into the specified register or register pair.

SOURCE CODE	OBJECT CODE
ld a,(ADDR)	3A DR AD
ld a,(bc)	0A
ld a,(de)	1A
ld a,(hl)	7E
ld a,(ix+DIS)	DD 7E SIA
ld a,(iy+DIS)	FD 7E DIS

SOURCE CODE	OBJECT CODE	
ld a,a	7F	
ld a,b	78	
ld a,c	79	
ld a,d	7A	
ld a,NN	3E NN	
ld a,e	7B	
ld a,h	7C	
ld a,l	7D	
ld a,i	ED 57	— interrupt vector
ld a,r	ED 5F	— refresh register
ld b,(hl)	46	
ld b,(ix+DIS)	DD 46 DIS	
ld b,(iy+DIS)	FD 46 DIS	
ld b,a	47	
ld b,b	40	
ld b,c	41	
ld b,d	42	
ld b,NN	06 NN	
ld b,e	43	
ld b,h	44	
ld b,l	45	
ld bc,(ADDR)	ED 4B DR AD	
ld bc,HHLL	01 LL HH	
ld c,(hl)	4E	
ld c,(ix+DIS)	DD 4E DIS	
ld c,(iy+DIS)	FD 4E DIS	
ld c,a	4F	
ld c,b	48	
ld c,c	49	
ld c,d	4A	
ld c,NN	0E NN	
ld c,e	4B	
ld c,h	4C	
ld c,l	4D	

SOURCE CODE	OBJECT CODE	
ld d,(hl)	56	
ld d,(ix+DIS)	DD 56 DIS	
ld d,(iy+DIS)	FD 56 DIS	
ld d,a	57	
ld d,b	50	
ld d,c	51	
ld d,d	52	
ld d,NN	16 NN	
ld d,e	53	
ld d,h	54	
ld d,l	55	
ld de,(ADDR)	ED 5B DR AD	
ld de,HHLL	11 LL HH	
ld e,(hl)	5E	
ld e,(ix+DIS)	DD 5E DIS	
ld e,(iy+DIS)	FD 5E DIS	
ld e,a	5F	
ld e,b	58	
ld e,c	59	
ld e,d	5A	
ld e,NN	1E NN	
ld e,e	5B	
ld e,h	5C	
ld e,l	5D	
ld h,(hl)	66	
ld h,(ix+DIS)	DD 66 DIS	
ld h,(iy+DIS)	FD 66 DIS	
ld h,a	67	
ld h,b	60	
ld h,c	61	
ld h,d	62	
ld h,NN	26 NN	
ld h,e	63	
ld h,h	64	
ld h,l	65	
ld hl,(ADDR)	ED 6B DR AD	
ld hl,(ADDR)	2A DR AD	
ld hl,HHLL	21 LL HH	
ld i,a	ED 47	— interrupt vector
ld r,a	ED 4F	— refresh register

SOURCE CODE	OBJECT CODE
ld ix,(ADDR)	DD 2A DR AD
ld ix,HHLL	DD 21 LL HH
ld iy,(ADDR)	FD 2A DR AD
ld iy,HHLL	FD 21 LL HH
ld l,(hl)	6E
ld l,(ix+DIS)	DD 6E DIS
ld l,(iy+DIS)	FD 6E DIS
ld l,a	6F
ld l,b	68
ld l,c	69
ld l,d	6A
ld l,NN	2E NN
ld l,e	6B
ld l,h	6C
ld l,l	6D
ld sp,(ADDR)	ED 7B DR AD
ld sp,HHLL	32 LL HH
ld sp,hl	F9
ld sp,ix	DD F9
ld sp,iy	FD F9

****

Block move instructions

The "hl" register pair points to the start address of the block of data to be moved. The register pair "de" points to the first byte of the destination memory area. The register pair "bc" contains the number of bytes to be moved.

The byte counter (bc) is decremented each time a byte is copied from the source byte (pointed by "hl") to the destination byte (pointed by "de"). Pointers "hl" and "de" are incremented or decremented according to which instruction is used.

If the repeat instruction is used then the operation will continue repeating until the byte count is zero.

SOURCE CODE	OBJECT CODE	
ldd	ED A8	— hl and de decrement
lddr	ED B8	— as ldd with repeat
ldi	ED A0	— hl and de increment
ldir	ED B0	— as ldi with repeat

Logical "or" instructions

These instructions perform a logical "or" operation between the stated data (value, register, or memory byte contents) and the "a" register. The result is placed into the "a" register.

The "or" instruction performs a bitwise comparison between two 8 bit numbers — the corresponding bit in the result number is set as follows:

1) both compared bits = 0 then result bit = 0.

2) any other condition then result bit = 1.

SOURCE CODE	OBJECT CODE
or (hl)	B6
or (ix+DIS)	DD B6 DIS
or (iy+DIS)	FD B6 DIS
or a	B7
or b	B0
or c	B1
or d	B2
or NN	F6 NN
or e	B3
or h	B4
or l	B5

****

Logical "xor" instructions

These instructions work in the same way as the "or" instructions but the results are as follows:

1) both compared bits the same then result bit = 0.

2) compared bits different then result bit = 1.

SOURCE CODE	OBJECT CODE
xor (hl)	AE
xor (ix+DIS)	DD AE DIS
xor (iy+DIS)	FD AE DIS
xor a	AF
xor b	A8
xor c	A9
xor d	AA
xor NN	EE NN
xor e	AB
xor h	AC
xor l	AD

Output instructions

The "out" instructions transfer data through a specified output port. The output port number is usually specified in register "c" and the data is contained in the specified register.

SOURCE CODE	OBJECT CODE
out (c),a	ED 79
out (c),b	ED 41
out (c),c	ED 49
out (c),d	ED 51
out (c),e	ED 59
out (c),h	ED 61
out (c),l	ED 69
out (PORT),a	D3 PORT

****

Block output instructions

The required output port number is placed into register "c". The start address of the block of memory to be output is placed into the "hl" register pair. The "b" register is used as a counter which decrements as each byte is output. The auto repeat instructions will terminate when the "b" register counts down to zero.

SOURCE CODE	OBJECT CODE	
outd	ED AB	— hl pointer decrements
otdr	ED BB	— as outd with repeat
outi	ED A3	— hl pointer increments
otir	ED B3	— as outi with repeat

****

Stack operations (push)

The 16 bit contents of a register pair is pushed onto the stack and the stack pointer is decremented by two. The low byte of the 16 bit number is pushed into the address stack pointer minus 2 and the high byte goes into the address stack pointer minus 1. NOTE that registers "a" and "f" act like a standard register pair for stack operations.

SOURCE CODE	OBJECT CODE
push af	F5
push bc	C5
push de	D5
push hl	E5
push ix	DD E5
push iy	FD E5

****

Stack operations (pop)

A 16 bit value is popped off the stack into the specified register pair and the stack pointer is incremented by two. Values need not be popped into the registers from which they were originally pushed and so push and pop are often used simply to tranfer data from one register to another.

SOURCE CODE	OBJECT CODE
pop af	F1
pop bc	C1
pop de	D1
pop hl	E1
pop ix	DD E1
pop iy	FD E1

****

The bit reset instructions

The specified bit in the specified register or memory location is reset to zero.

SOURCE CODE	OBJECT CODE
res .0,(hl)	CB 86
res .0,(ix+DIS)	DD CB DIS 86
res .0,(iy+DIS)	FD CB DIS 86
res .0,a	CB 87
res .0,b	CB 80
res .0,c	CB 81
res .0,d	CB 82
res .0,e	CB 83
res .0,h	CB 84
res .0,l	CB 85

SOURCE CODE	OBJECT CODE
res .1,(hl)	CB 8E
res .1,(ix+DIS)	DD CB DIS 8E
res .1,(iy+DIS)	FD CB DIS 8E
res .1,a	CB 8F
res .1,b	CB 88
res .1,c	CB 89
res .1,d	CB 8A
res .1,e	CB 8B
res .1,h	CB 8C
res .1,l	CB 8D
res .2,(hl)	CB 96
res .2,(ix+DIS)	DD CB DIS 96
res .2,(iy+DIS)	FD CB DIS 96
res .2,a	CB 97
res .2,b	CB 90
res .2,c	CB 91
res .2,d	CB 92
res .2,e	CB 93
res .2,h	CB 94
res .2,l	CB 95
res .3,(hl)	CB 9E
res .3,(ix+DIS)	DD CB DIS 9E
res .3,(iy+DIS)	FD CB DIS 9E
res .3,a	CB 9F
res .3,b	CB 98
res .3,c	CB 99
res .3,d	CB 9A
res .3,e	CB 9B
res .3,h	CB 9C
res .3,l	CB 9D
res .4,(hl)	CB A6
res .4,(ix+DIS)	DD CB DIS A6
res .4,(iy+DIS)	FD CB DIS A6
res .4,a	CB A7
res .4,b	CB A0
res .4,c	CB A1
res .4,d	CB A2
res .4,e	CB A3
res .4,h	CB A4
res .4,l	CB A5

SOURCE CODE	OBJECT CODE
res .5,(hl)	CB AE
res .5,(ix+DIS)	DD CB DIS AE
res .5,(iy+DIS)	FD CB DIS AE
res .5,a	CB AF
res .5,b	CB A8
res .5,c	CB A9
res .5,d	CB AA
res .5,e	CB AB
res .5,h	CB AC
res .5,l	CB AD
res .6,(hl)	CB B6
res .6,(ix+DIS)	DD CB DIS B6
res .6,(iy+DIS)	FD CB DIS B6
res .6,a	CB B7
res .6,b	CB B0
res .6,c	CB B1
res .6,d	CB B2
res .6,e	CB B3
res .6,h	CB B4
res .6,l	CB B5
res .7,(hl)	CB BE
res .7,(ix+DIS)	DD CB DIS BE
res .7,(iy+DIS)	FD CB DIS BE
res .7,a	CB BF
res .7,b	CB B8
res .7,c	CB B9
res .7,d	CB BA
res .7,e	CB BB
res .7,h	CB BC
res .7,l	CB BD

****

The bit set instructions

The specified bit in the specified register or memory location is set to one.

SOURCE CODE	OBJECT CODE
set .0,(hl)	CB C6
set .0,(ix+DIS)	DD CB DIS C6
set .0,(iy+DIS)	FD CB DIS C6
set .0,a	CB C7
set .0,b	CB C0
set .0,c	CB C1
set .0,d	CB C2
set .0,e	CB C3
set .0,h	CB C4
set .0,l	CB C5
set .1,(hl)	CB CE
set .1,(ix+DIS)	DD CB DIS CE
set .1,(iy+DIS)	FD CB DIS CE
set .1,a	CB CF
set .1,b	CB C8
set .1,c	CB C9
set .1,d	CB CA
set .1,e	CB CB
set .1,h	CB CC
set .1,l	CB CD
set .2,(hl)	CB D6
set .2,(ix+DIS)	DD CB DIS D6
set .2,(iy+DIS)	FD CB DIS D6
set .2,a	CB D7
set .2,b	CB D0
set .2,c	CB D1
set .2,d	CB D2
set .2,e	CB D3
set .2,h	CB D4
set .2,l	CB D5
set .3,(hl)	CB DE
set .3,(ix+DIS)	DD CB DIS DE
set .3,(iy+DIS)	FD CB DIS DE
set .3,a	CB DF
set .3,b	CB D8
set .3,c	CB D9
set .3,d	CB DA
set .3,e	CB DB
set .3,h	CB DC
set .3,l	CB DD

SOURCE CODE	OBJECT CODE
set .4,(hl)	CB E6
set .4,(ix+DIS)	DD CB DIS E6
set .4,(iy+DIS)	FD CB DIS E6
set .4,a	CB E7
set .4,b	CB E0
set .4,c	CB E1
set .4,d	CB E2
set .4,e	CB E3
set .4,h	CB E4
set .4,l	CB E5
set .5,(hl)	CB EE
set .5,(ix+DIS)	DD CB DIS EE
set .5,(iy+DIS)	FD CB DIS EE
set .5,a	CB EF
set .5,b	CB E8
set .5,c	CB E9
set .5,d	CB EA
set .5,e	CB EB
set .5,h	CB EC
set .5,l	CB ED
set .6,(hl)	CB F6
set .6,(ix+DIS)	DD CB DIS F6
set .6,(iy+DIS)	FD CB DIS F6
set .6,a	CB F7
set .6,b	CB F0
set .6,c	CB F1
set .6,d	CB F2
set .6,e	CB F3
set .6,h	CB F4
set .6,l	CB F5
set .7,(hl)	CB FE
set .7,(ix+DIS)	DD CB DIS FE
set .7,(iy+DIS)	FD CB DIS FE
set .7,a	CB FF
set .7,b	CB F8
set .7,c	CB F9
set .7,d	CB FA
set .7,e	CB FB
set .7,h	CB FC
set .7,l	CB FD

The return instructions

The return address is popped off the stack into the program counter and the operation continues from that address. Conditional returns are subject to the condition being fulfilled.

SOURCE CODE	OBJECT CODE	
ret	C9	— unconditional
ret c	D8	— if carry flag is set
ret m	F8	— if sign flag is set
ret nc	D0	— if carry flag is reset
ret nz	C0	— if zero flag is reset
ret p	F0	— if sign flag is reset
ret pe	E8	— if parity flag is set
ret po	E0	— if parity flag is reset
ret z	C8	— if zero flag is set
reti	ED 4D	— return from an interrupt service routine
retn	ED 45	— return from a non maskable interrupt service routine

****

Restart instructions

The Z80 restarts provide single byte instructions to jump to certain frequently used ROM routines in page 0. The application on the MSX is given for each restart.

SOURCE CODE	OBJECT CODE	
rst .00	C7	— reboot computer
rst .08	CF	— basic syntax check
rst .10	D7	— get next basic character
rst .18	DF	— print character in "a"
rst .20	E7	— compares "hl" and "de"
rst .28	EF	— performs interslot call
rst .30	F7	— checks type of FAC
rst .38	FF	— interrupt routine

****

The rotate instructions

Rotate left

The bits in the specified register or memory location are moved one to the left. Bit 7 moves into the carry flag and the previous contents of the carry flag move into bit 0.

SOURCE CODE	OBJECT CODE
rl (hl)	CB 16
rl (ix+DIS)	DD CB DIS 16
rl (iy+DIS)	FD CB DIS 16
rl a	CB 17
rla	17
rl b	CB 10
rl c	CB 11
rl d	CB 12
rl e	CB 13
rl h	CB 14
rl l	CB 15

Rotate left with carry

The bits in the specified register or memory location are moved one to the left. Bit 7 moves into the carry flag and is copied into bit 0.

SOURCE CODE	OBJECT CODE
rlc (hl)	CB 06
rlc (ix+DIS)	DD CB DIS 06
rlc (iy+DIS)	FD CB DIS 06
rlc a	CB 07
rlca	07
rlc b	CB 00
rlc c	CB 01
rlc d	CB 02
rlc e	CB 03
rlc h	CB 04
rlc l	CB 05

Rotate right

The bits in the specified register or memory location are moved one to the right. Bit 0 moves into the carry flag and the previous contents of the carry flag move into bit 7.

SOURCE CODE	OBJECT CODE
rr (hl)	CB 1E
rr (ix+DIS)	DD CB DIS 1E
rr (iy+DIS)	FD CB DIS 1E
rr a	CB 1F
rra	1F
rr b	CB 18
rr c	CB 19
rr d	CB 1A
rr e	CB 1B
rr h	CB 1C
rr l	CB 1D

Rotate right with carry

The bits in the specified register or memory location are moved one to the right. Bit 0 moves into the carry flag and is copied into bit 7.

SOURCE CODE	OBJECT CODE
rrc (hl)	CB 0E
rrc (ix+DIS)	DD CB DIS 0E
rrc (iy+DIS)	FD CB DIS 0E
rrc a	CB 0F
rrca	0F
rrc b	CB 08
rrc c	CB 09
rrc d	CB 0A
rrc e	CB 0B
rrc h	CB 0C
rrc l	CB 0D

****

Two special rotate instructions

These instructions operate on the memory byte pointed to by "hl" and the "a" register.

SOURCE CODE	OBJECT CODE
rld	ED 6F

The following operations take place:

1) The lower 4 bits in (hl) move into the upper 4 bits.

2) The upper 4 bits in (hl) move into the lower 4 bits of "a"

3) The lower 4 bits in "a" move into the lower 4 bits in (hl)

This instruction can be used to multiply the contents of a memory byte by 16.

SOURCE CODE	OBJECT CODE
rrd	ED 67

The following operations take place:

1) The lower 4 bits in (hl) move into the lower 4 bits of "a"
2) The upper 4 bits in (hl) move into the lower 4 bits.
3) The lower 4 bits in "a" move into the upper 4 bits in (hl)

This instruction can be used to divide the contents of a memory byte by 16.

****

The shift instructions

Shift left arithmetic

The bits in a register or memory location are shifted one to the left. Bit 7 moves into the carry flag and bit 0 is reset to zero.

SOURCE CODE	OBJECT CODE
sla (hl)	CB 26
sla (ix+DIS)	DD CB DIS 26
sla (iy+DIS)	FD CB DIS 26
sla a	CB 27
sla b	CB 20
sla c	CB 21
sla d	CB 22
sla e	CB 23
sla h	CB 24
sla l	CB 25

Shift right arithmetic

The bits in a register or memory location are shifted one to the right. Bit 0 moves into the carry flag and bit 7 remains unchanged.

SOURCE CODE	OBJECT CODE
sra (hl)	CB 2E
sra (ix+DIS)	DD CB DIS 2E
sra (iy+DIS)	FD CB DIS 2E
sra a	CB 2F
sra b	CB 28
sra c	CB 29
sra d	CB 2A
sra e	CB 2B
sra h	CB 2C
sra l	CB 2D

Shift right logical

The bits in a register or memory location are shifted one to the right. Bit 0 moves into the carry flag and bit 7 is reset to zero.

SOURCE CODE	OBJECT CODE
srl (hl)	CB 3E
srl (ix+DIS)	DD CB DIS 3E
srl (iy+DIS)	FD CB DIS 3E
srl a	CB 3F
srl b	CB 38
srl c	CB 39
srl d	CB 3A
srl e	CB 3B
srl h	CB 3C
srl l	CB 3D

****

Subtract instructions

Subtract without carry

The specified value (actual value, register, or memory location contents) is subtracted from the contents of the "a" register and the result is placed into the "a" register.

SOURCE CODE	OBJECT CODE
sub (hl)	96
sub (ix+DIS)	DD 96 DIS
sub (iy+DIS)	FD 96 DIS
sub a	97
sub b	90
sub c	91
sub d	92
sub NN	D6 NN
sub e	93
sub h	94
sub l	95

****

Subtract with carry

The contents of the carry flag plus the specified value (actual value, register, or memory location contents) are subtracted from the contents of the "a" register and the result is placed into the "a" register.

SOURCE CODE	OBJECT CODE
sbc (hl)	9E
sbc (ix+DIS)	DD 9E DIS
sbc (iy+DIS)	FD 9E DIS
sbc a	9F
sbc b	98
sbc c	99
sbc d	9A
sbc NN	DE NN
sbc e	9B
sbc h	9C
sbc l	9D

****

16 Bit subtract with carry

The carry flag is subtracted from the contents of the HL register pair. The contents of the second named register pair are then subtracted from the contents of HL. The result is stored in HL.

SOURCE CODE	OBJECT CODE
sbc hl,bc	ED 42
sbc hl,de	ED 52
sbc hl,hl	ED 62
sbc hl,sp	ED 72

Special relative jump

The B register is used as a counter. The B register contents are decremented each time the instruction executes and if B is greater than zero the relative jump occurs.

SOURCE CODE	OBJECT CODE
djnz,DIS	10 DIS

APPENDIX 2

MSX BASIC WORD ROM ROUTINES (version 1.0)

BASIC WORD	TOKEN (hex)	ROM ADDRESS (hex)
END	81	63EA
FOR	82	4524
NEXT	83	6527
DATA	84	485B
INPUT	85	4B6C
DIM	86	5E9F
READ	87	4B9F
LET	88	4880
GOTO	89	47E8
RUN	8A	479E
IF	8B	49E5
RESTORE	8C	63C9
GOSUB	8D	47B2
RETURN	8E	4821
REM	8F	485D
STOP	90	63E3
PRINT	91	4A24
CLEAR	92	64AF
LIST	93	522E
NEW	94	6286
ON	95	48E4
WAIT	96	401C
DEF	97	501D
POKE	98	5423
CONT	99	6424
CSAVE	9A	6FB7
CLOAD	9B	703F
OUT	9C	4016
LPRINT	9D	4A1D
LLIST	9E	5229
CLS	9F	00C3
WIDTH	A0	51C9
ELSE	A1	485D
TRON	A2	6438
TROFF	A3	6439
SWAP	A4	643E
ERASE	A5	6477
ERROR	A6	49AA

MSX BASIC WORD ROM ROUTINES (version 1.0 Cont.)

BASIC WORD	TOKEN (hex)	ROM ADDRESS (hex)
RESUME	A7	495D
DELETE	A8	53E2
AUTO	A9	49B5
RENUM	AA	5468
DEFSTR	AB	4718
DEFINT	AC	471B
DEFSNG	AD	471E
DEFDBL	AE	4721
LINE	AF	4B0E
OPEN	B0	6AB7
FIELD	B1	7C52
GET	B2	775B
PUT	B3	7758
CLOSE	B4	6C14
LOAD	B5	6B5D
MERGE	B6	6B5E
FILES	B7	6C2F
LSET	B8	7C48
RSET	B9	7C4D
SAVE	BA	6BA3
LFILES	BB	6C2A
CIRCLE	BC	5B11
COLOR	BD	7980
DRAW	BE	5D6E
PAINT	BF	59C5
BEEP	C0	00C0
PLAY	C1	73E5
PSET	C2	57EA
PRESET	C3	57E5
SOUND	C4	73CA
SCREEN	C5	79CC
VPOKE	C6	7BE2
SPRITE	C7	7A48
VDP	C8	7B37
BASE	C9	7B5A
CALL	CA	55A8
TIME	CB	7911
KEY	CC	786C
MAX	CD	7E4B
MOTOR	CE	73B7

BASIC WORD	TOKEN (hex)	ROM ADDRESS (hex)
BLOAD	CF	6EC6
BSAVE	D0	6E92
DSKO$	D1	7C16
SET	D2	7C1B
NAME	D3	7C20
KILL	D4	7C25
IPL	D5	7C2A
COPY	D6	7C2F
CMD	D7	7C34
LOCATE	D8	7766
USR	DD	4FD5
FN	DE	5040
ERL	E2	4E0B
STRING$	E3	6829
INSTR	E5	68EB
VARPTR	E7	4E41
CSRLIN	E8	790A
ATTR$	E9	7C43
DSKI$	EA	7C3E
INKEY$	EC	7347
POINT	ED	5803
LEFT$	FF 81	6861
RIGHT$	FF 82	6891
MID$	FF 83	689A
SGN	FF 84	2E97
INT	FF 85	30CF
ABS	FF 86	2E82
SQR	FF 87	2AFF
RND	FF 88	2BDF
SIN	FF 89	29AC
LOG	FF 8A	2A72
EXP	FF 8B	2B4A
COS	FF 8C	2993
TAN	FF 8D	29FB
ATN	FF 8E	2A14
FRE	FF 8F	69F2
INP	FF 90	4001
POS	FF 91	4FCC
LEN	FF 92	67FF
STR$	FF 93	6604

MSX BASIC WORD ROM ROUTINES (version 1.0 Cont.)

BASIC WORD	TOKEN (hex)	ROM ADDRESS (hex)
VAL	FF 94	68BB
ASC	FF 95	680B
CHR$	FF 96	681B
PEEK	FF 97	541C
VPEEK	FF 98	7BF5
SPACE$	FF 99	6848
OCT$	FF 9A	65F5
HEX$	FF 9B	65FA
LPOS	FF 9C	4FC7
BIN$	FF 9D	65FF
CINT	FF 9E	2F8A
CSNG	FF 9F	2FB2
CDBL	FF A0	303A
FIX	FF A1	30BE
STICK	FF A2	7940
STRIG	FF A3	794C
PDL	FF A4	795A
PAD	FF A5	7969
DSKF	FF A6	7C39
FPOS	FF A7	6D39
CVI	FF A8	7C66
CVS	FF A9	7C6B
CVD	FF AA	7C70
EOF	FF AB	6D25
LOC	FF AC	6D03
LOF	FF AD	6D14
MKI$	FF AE	7C57
MKS$	FF AF	7C5C
MKD$	FF B0	7C61

Certain ROM routines operate without any extra parameters — eg CLS or STOP. To use these routines you simply CALL the ROM address from your MC program.

The following source file illustrates the source code to clear the screen using the basic word CLS:

SOURCE FILE APPENDIX 2.1

```
10 REM [ .d000
20 REM call .c3
30 REM ret
40 REM ]
```

Most ROM routines need input parameters – eg LOCATE 10,1 or PRINT "MSX". To use these routines you must point the HL register pair to the address of a machine code string which contains the parameters, and then CALL the routine. The MC string should not contain the BASIC word itself but may contain other basic words in tokenised form.

The following source file illustrates the source code required to perform the basic command COLOR 1,11,8:

SOURCE FILE APPENDIX 2.2

```
10 REM [.d000
20 REM ld hl,Str
30 REM call .7980
40 REM ret
50 REM Str
60 REM $1,11,8:
70 REM ]
```

HINT

To get the correct tokenised form, write the instruction into a basic line and peek out the tokens.

APPENDIX 3

MORE ROM ROUTINES

In appendix 2 the positions of the BASIC WORD MACRO ROUTINES were given — in this appendix some of the more useful PRIMITIVE ROUTINES are given.

ERAFNK

ADDRESS	&H00cc
ENTRY	NONE
EFFECT	ERASES THE FUNCTION KEY DISPLAY

DSPFNK

ADDRESS	&H00cf
ENTRY	NONE
EFFECT	DISPLAYS THE FUNCTION KEY DEFINITIONS

FNKSB

ADDRESS	&H00c9
ENTRY	NONE
EFFECT	CHECKS IF FUNCTION KEY DISPLAY IS ACTIVE AND IF SO DISPLAYS THE KEYS OTHERWISE DOES NOTHING

RSTFNK

ADDRESS	&H003e
ENTRY	NONE
EFFECT	RESTORES THE FUNCTION KEYS TO DEFAULT STRINGS

DISSCR

ADDRESS	&H0041
ENTRY	NONE
EFFECT	DISABLES THE SCREEN DISPLAY

ENASCR

ADDRESS	&H0044
ENTRY	NONE
EFFECT	ENABLES THE SCREEN DISPLAY

WRTVDP

ADDRESS	&H0047
ENTRY	VDP REGISTER NUMBER IN C DATA IN B
EFFECT	WRITES DATA TO VDP REGISTER

RDVDP

ADDRESS	&H013e
ENTRY	NONE
EFFECT	RETURNS VDP STATUS REGISTER CONTENTS IN A

RDVRM

ADDRESS	&H004a
ENTRY	VRAM ADDRESS IN HL
EFFECT	READS VRAM DATA INTO A

WRTVRAM

ADDRESS	&H004d
ENTRY	VRAM ADDRESS IN HL DATA IN A
EFFECT	WRITES DATA TO VRAM BYTE

FILVRAM

ADDRESS	&H0056
ENTRY	VRAM ADDRESS IN HL NUMBER OF BYTES IN BC DATA IN A
EFFECT	FILLS THE BLOCK OF MEMORY FROM ADDRESS HL FOR BC BYTES WITH THE DATA IN A

LDIRMV

ADDRESS	&H0059
ENTRY	VRAM SOURCE ADDRESS IN HL RAM DESTINATION ADDRESS IN DE NUMBER OF BYTES IN BC
EFFECT	MOVES A BLOCK OF VRAM MEMORY INTO NORMAL RAM

LDIRVM

ADDRESS	&H005c
ENTRY	RAM SOURCE ADDRESS IN HL VRAM DESTINATION ADDRESS IN DE NUMBER OF BYTES IN BC
EFFECT	MOVES A BLOCK OF RAM MEMORY INTO VRAM

MAPXYC

ADDRESS	&H0111
ENTRY	X CO-ORDINATE IN BC REGISTER Y CO-ORDINATE IN DE REGISTER
EFFECT	POSITIONS THE GRAPHICS POINTER TO (X,Y)

SETC

ADDRESS	&H0120
ENTRY	GRAPHICS POINTER LOCATED AT (X,Y) REQUIRED COLOR IN ATRBYT (&Hf3f2)
EFFECT	SETS THE PIXEL (X,Y) TO COLOR IN ATRBYT

SETATR

ADDRESS	&H011a
ENTRY	COLOR NUMBER IN THE A REGISTER
EFFECT	SET ATRBYT TO THE COLOR IN 'A'

READC

ADDRESS	&H011d
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	READ COLOR OF PIXEL (X,Y) INTO 'A'

RIGHTC

ADDRESS	&H00fc
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X+1,Y)

LEFTC

ADDRESS	&H00ff
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X-1,Y)

UPC

ADDRESS	&H0102
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X,Y−1)

TUPC

ADDRESS	&H0105
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	SETS CARRY FLAG AND RETURNS IF TOP OF SCREEN IS REACHED ELSE SAME AS UPC

DOWNC

ADDRESS	&H0108
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X,Y+1)

TDOWNC

ADDRESS	&H010b
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	SETS CARRY FLAG AND RETURNS IF BOTTOM OF SCREEN IS REACHED ELSE SAME AS DOWNC

CHGCLR

ADDRESS	&H0062
ENTRY	REQUIRED FOREGROUND COLOR IN &Hf3e9 REQUIRED BACKGROUND COLOR IN &Hf3ea REQUIRED BORDER COLOR IN &Hf3eb
EFFECT	CHANGES THE SCREEN COLORS

CLRSPR

ADDRESS	&H0069
ENTRY	SCREEN NUMBER IN SCRMOD (&Hfcaf)
EFFECT	FILLS SPRITE PATTERN AREA WITH 0 SETS SPRITE NUMBERS IN ATTRIBUTE TABLE TO PLANE NUMBERS SETS SPRITE COLORS TO FOREGROUND COLOR SETS SPRITE VERTICAL POSITIONS TO 209

INITXT

ADDRESS	&H006c
ENTRY	BASE ADDRESS OF THE TEXT NAME TABLE IN &Hf3b3 and &Hf3b4 BASE ADDRESS OF THE TEXT PATTERN TABLE IN &Hf3b7 and &Hf3b8
EFFECT	INITIALISES THE SCREEN TO TEXT MODE SCREEN 0

INIT32

ADDRESS	&H006f
ENTRY	BASE ADDRESS OF NAME TABLE IN &Hf3bd and &Hf3be BASE ADDRESS OF COLOR TABLE IN &Hf3bf and &Hf3c0 BASE ADDRESS OF PATTERN TABLE IN&Hf3c1 and &Hf3c2 BASE ADDRESS OF SPRITE ATTRIBUTE TABLE IN &Hf3c3 and &Hf3c4 BASE ADDRESS OF SPRITE PATTERN TABLE IN &Hf3c5 and &Hf3c6
EFFECT	INITIALISES THE SCREEN TO TEXT MODE SCREEN 1

INITGRP

ADDRESS	&H0072
ENTRY	BASE ADDRESS OF NAME TABLE IN &Hf3c7 and &Hf3c8 BASE ADDRESS OF COLOR TABLE IN &Hf3c9 and &Hf3ca BASE ADDRESS OF PATTERN TABLE IN &Hf3cb and &Hf3cc BASE ADDRESS OF SPRITE ATTRIBUTE TABLE IN &Hf3cd and &Hf3ce BASE ADDRESS OF SPRITE PATTERN TABLE IN &Hf3cf and &Hf3d0
EFFECT	INITIALISES THE SCREEN TO GRAPHICS MODE SCREEN 2

INITMLT

ADDRESS	&H0075
ENTRY	BASE ADDRESS OF NAME TABLE IN &Hf3d1 and &Hf3d2 BASE ADDRESS OF COLOR TABLE IN &Hf3d3 and &Hf3d4 BASE ADDRESS OF PATTERN TABLE IN &Hf3d5 and &Hf3d6 BASE ADDRESS OF SPRITE ATTRIBUTE TABLE IN &Hf3d7 and &Hf3d8 BASE ADDRESS OF SPRITE PATTERN TABLE IN &Hf3d9 and &Hf3da
EFFECT	INITIALISES THE SCREEN TO GRAPHICS MODE SCREEN 3

CALPAT

ADDRESS	&H0084
ENTRY	SPRITE NUMBER IN A
EFFECT	RETURNS THE ADDRESS OF THE SPRITE PATTERN IN HL

CALATR

ADDRESS	&H0087
ENTRY	SPRITE NUMBER IN A
EFFECT	RETURNS SPRITE ATTRIBUTE ADDRESS IN HL

GSPSIZ

ADDRESS	&H008a
ENTRY	NONE
EFFECT	RETURNS NUMBER OF BYTES IN SPRITE DEFINITION IN THE A REGISTER. CARRY FLAG SET IF SPRITES ARE 16 * 16 AND RESET IF SPRITES ARE 8 * 8

GRPPRT

ADDRESS	&H008d
ENTRY	CHARACTER CODE IN A
EFFECT	PRINTS THE CHARACTER ON THE GRAPHICS SCREEN

WRTPSG

ADDRESS	&H0093
ENTRY	PSG REGISTER NUMBER IN A DATA IN E
EFFECT	WRITES DATA TO PSG REGISTER

RDPSG

ADDRESS	&H0096
ENTRY	PSG REGISTER NUMBER IN A
EFFECT	RETURNS DATA FROM PSG REGISTER IN A

CHSNS

ADDRESS	&H009c
ENTRY	NONE
EFFECT	RESETS ZERO FLAG IF THERE IS A CHARACTER IN THE KEYBOARD BUFFER

CHGET

ADDRESS	&H009f
ENTRY	NONE
EFFECT	WAITS FOR A CHARACTER TO BE TYPED AND RETURNS WITH THE CHARACTER CODE IN A

POSIT

ADDRESS	00c6
ENTRY	SCREEN COLUMN IN H SCREEN ROW IN L
EFFECT	LOCATES CURSOR AT ROW L COLUMN H

CHPUT

ADDRESS	&H00a2
ENTRY	CHARACTER CODE IN A
EFFECT	PRINTS CHARACTER ON SCREEN

SNSMAT

ADDRESS	&H0141
ENTRY	KEYBOARD MATRIX ROW IN A
EFFECT	RETURNS STATUS OF ROW IN A (SEE APPENDIX 8)

LPTOUT

ADDRESS	&H00a5
ENTRY	CHARACTER CODE IN REGISTER 'A'
EFFECT	OUTPUTS A CHARACTER TO THE LINE PRINTER. THE CARRY FLAG IS SET IF THE OUTPUT IS ABORTED

LPTSTT

ADDRESS	&H00a8
ENTRY	NONE
EFFECT	CHECKS THE LINE PRINTER STATUS AND RETURNS: 255 IN 'A' and ZERO FLAG RESET IF PRINTER READY ZERO IN 'A' and ZERO FLAG SET IF PRINTER NOT READY

GTSTCK

ADDRESS	&H00d5
ENTRY	JOYSTICK IDENTITY IN 'A' REGISTER
EFFECT	RETURNS THE JOYSTICK DIRECTION IN THE 'A' REGISTER. JOYSTICK IDENTITY IS 1 FOR JOYSTICK 1 AND 2 FOR JOYSTICK 2

GTTRIG

ADDRESS	&H00d8
ENTRY	TRIGGER BUTTON IDENTITY IN 'A' REGISTER
EFFECT	RETURNS THE STATUS OF THE TRIGGER BUTTON IN THE 'A' REGISTER. 255 IF PRESSED AND ZERO IF NOT PRESSED. TRIGGER IDENTITY IS 1 AND 3 FOR TRIGGERS ON JOYSTICK 1 – 2 AND 4 FOR TRIGGERS ON JOYSTICK 2

KILBUF

ADDRESS	&H0156
ENTRY	NONE
EFFECT	CLEARS THE KEYBOARD BUFFER

CHGCAP

ADDRESS	&H132
ENTRY	ZERO IN THE 'A' REGISTER TO TURN THE CAPS LAMP ON. NON ZERO IN 'A' TO TURN THE LAMP OFF
EFFECT	CHANGES THE STATUS OF THE CAPS LAMP

TAPION

ADDRESS	&H00e1
ENTRY	NONE
EFFECT	STARTS TAPE AND READS LEAD IN HEADER. SETS CARRY FLAG IF ABORTED

TAPIN

ADDRESS	&H00e4
ENTRY	NONE
EFFECT	READS BYTE FROM TAPE INTO REGISTER A. CARRY FLAG SET IF ABORTED

TAPIOF

ADDRESS	&H00e7
ENTRY	NONE
EFFECT	STOPS READING THE TAPE

TAPDON

ADDRESS	&H00ea
ENTRY	A = NON ZERO IF LONG HEADER REQUIRED A = 0 IF SHORT HEADER REQUIRED
EFFECT	STARTS TAPE AND WRITES HEADER TO IT

TAPOUT

ADDRESS	&H00ed
ENTRY	DATA IN REGISTER A
EFFECT	WRITES DATA TO TAPE

MORE ROM ROUTINES (Cont.)

TAPOOF

ADDRESS	&H00f0
ENTRY	NONE
EFFECT	STOPS WRITING TO THE TAPE

STMOTR

ADDRESS	&H00f3
ENTRY	A = 0 TO STOP TAPE MOTOR A = 1 TO START TAPE MOTOR A = 255 TO CHANGE TAPE MOTOR STATUS
EFFECT	SETS TAPE MOTOR

MORE TAPE INFORMATION

There are two types of header:

A LONG HEADER WITH A LENGTH OF 16 UNITS.

A SHORT HEADER WITH A LENGTH OF 4 UNITS.

The header contains no specific data but the computer sets the tape read baud rate according to the frequency of the header:

1200 BAUD HAS A FREQUENCY OF 2400 Hz.

2400 BAUD HAS A FREQUENCY OF 4800 Hz.

TAPE DATA

The data format is a start bit (0), eight data bits and then two stop bits (1). The order of data bits is from the least significant bit to the most significant bit.

FILE FORMATS

CSAVE FILE

LONG HEADER
10 * D3 hex
FILE NAME (6 characters)

SHORT HEADER
BASIC PROGRAM IN TOKENISED FORM
7 * 00 hex

FILE FORMATS (Cont.)

SAVE FILE (ASCII)

LONG HEADER
10 * EA hex
FILE NAME (6 characters)

SHORT HEADER
256 DATA BYTES
SHORT HEADER
256 DATA BYTES
SHORT HEADER
256 DATA BYTES
: : : : : : : : : : : : :
: : : : : : : : : : : : :
: : : : : : : : : : : : :
SHORT HEADER
256 DATA BYTES (CONTROL Z MARKS THE END OF FILE)

BSAVE FILE

LONG HEADER
10 * D0 hex
FILE NAME (6 characters)

SHORT HEADER
START ADDRESS (2 bytes)
END ADDRESS (2 bytes)
RUN ADDRESS (2 bytes)
MACHINE LANGUAGE FILE

APPENDIX 4

SCREEN FORMATTING AND EDITING COMMANDS

Your MSX machine is equipped with several sets of powerful screen formatting and editing commands which can be used in basic or machine code to give complete control over the text screen. These commands fall into three categories:

1) The IMMEDIATE CONTROL CODES which can be entered directly at the keyboard and perform their function immediately.
2) The IMMEDIATE/PROGRAM CONTROL CODES which can be entered directly or incorporated into a program.
3) The ESCAPE SEQUENCES which can only be used within program code.

THE IMMEDIATE CONTROL CODES

These codes are obtained by pressing the CTRL key and the respective character key at the same time.

CODE	EFFECT
CTRL B	MOVE CURSOR TO THE START OF PREVIOUS WORD
CTRL E	CLEAR TEXT FROM CURSOR POSITION TO END OF LINE
CTRL F	MOVE CURSOR TO THE START OF THE NEXT WORD
CTRL H	BACKSPACE DELETING CHARACTER TO LEFT OF CURSOR
CTRL N	MOVE CURSOR TO THE END OF THE THE LINE
CTRL R	TOGGLES INSERT MODE
CTRL U	CLEAR ALL TEXT IN CURRENT LINE

THE IMMEDIATE/PROGRAM CONTROL CODES

These codes can be obtained immediately by pressing the CTRL key together with the respective character key. To use the codes within a program you must use the given program code.

THE IMMEDIATE/PROGRAM CONTROL CODES (Cont.)

CODE	PROGRAM CODE	EFFECT
CTRL G	PRINT CHR$(7)	SOUND THE BEEP
CTRL I	PRINT CHR$(9)	MOVE THE CURSOR TO THE NEXT TAB
CTRL K	PRINT CHR$(11)	MOVE THE CURSOR TO TOP LEFT (HOME)
CTRL L	PRINT CHR$(12)	CLEAR THE SCREEN AND HOME CURSOR
CTRL M	PRINT CHR$(13)	CARRIAGE RETURN
RIGHT	PRINT CHR$(28)	CURSOR RIGHT
LEFT	PRINT CHR$(29)	CURSOR LEFT
UP	PRINT CHR$(30)	CURSOR UP
DOWN	PRINT CHR$(31)	CURSOR DOWN

ESCAPE SEQUENCES

These sequences are obtained by entering the code sequence within your programs.

CODE SEQUENCE	EFFECT
PRINT CHR$(27)"j"	CLEAR THE SCREEN AND HOME THE CURSOR
PRINT CHR$(27)"E"	CLEAR THE SCREEN AND HOME THE CURSOR
PRINT CHR$(27)"K"	ERASE FROM CURSOR TO END OF LINE
PRINT CHR$(27)"J"	ERASE FROM CURSOR TO END OF PAGE
PRINT CHR$(27)"l"	ERASE ENTIRE LINE AT CURSOR POSITION
PRINT CHR$(27)"L"	INSERT A LINE AT CURSOR POSITION
PRINT CHR$(27)"M"	DELETE A LINE AT CURSOR POSITION
PRINT CHR$(27)"A"	CURSOR UP ONE LINE
PRINT CHR$(27)"B"	CURSOR DOWN ONE LINE
PRINT CHR$(27)"C"	CURSOR RIGHT ONE COLUMN
PRINT CHR$(27)"D"	CURSOR LEFT ONE COLUMN
PRINT CHR$(27)"H"	CURSOR HOME

APPENDIX 5

INPUT/OUTPUT PORT TABLE

Input and Output ports are the channels through which the Z80A microprocessor communicates with peripheral devices such as the screen or the printer. The Z80A is equipped with 256 input and 256 output ports — these ports are numbered from zero to &HFF.

A table of the functions of the various MSX STANDARD PORTS is given below. NOTE that the ports used by add on equipment (eg. disc drive or RS232 interface) are not given — these details are normally supplied with the equipment.

PORT NUMBER	I/O	DEVICE	DESCRIPTION
&H90	O	PRINTER	DATA STROBE (bit 0)
&H90	I	PRINTER	STATUS (bit 1 = 0 if ready)
&H91	O	PRINTER	DATA WRITE PORT
&H98	I	VDP	READ DATA
&H98	O	VDP	WRITE DATA
&H99	I	VDP	READ STATUS
&H99	O	VDP	COMMAND REGISTER
&HA0	O	PSG	REGISTER SELECT LATCH
&HA1	O	PSG	WRITE DATA
&HA2	I	PSG	READ DATA
&HA8	O	PPI	PORT A DATA WRITE
&HA8	I	PPI	PORT A DATA READ
&HA9	O	PPI	PORT B DATA WRITE
&HA9	I	PPI	PORT B DATA READ
&HAA	O	PPI	PORT C DATA WRITE
&HAA	I	PPI	PORT C DATA READ
&HAB	O	PPI	CONTROL WORD REGISTER

SPECIAL NOTE TO PROGRAMMERS — to maintain compatibility with future MSX versions you should always access the peripherals through the primitive I/O routines (in ROM) and never directly through the ports. The one exception to this rule is VDP access — the VDP read data port number will always be stored in ROM address &H0006 and the write data port number will always be stored in address &H0007. Your programs should collect the port number from these addresses. Any direct I/O routines given in this book are intended for information only.

APPENDIX 6

THE BASIC STATEMENT HANDLER (MSX BASIC version 1.0)

This ROM ROUTINE is used by every basic command and function to interpret the basic token and call the required execution routines. The statement handler is a very useful routine for the machine code programmer because it gives access to all basic routines in the ROM. The hl register pair is pointed to the start of the statement, the "a" register is loaded with the first character of the statement, and the routine is called at address &H4646.

To illustrate the use of the statement handler lets look at a short program to print the address of the stack pointer onto the screen. In basic the program looks like this:

```
10 PRINT HEX$(PEEK(&HF6B1)+&H100*PEEK(&HF6B2))
```

This routine in machine code uses the following source:

SOURCE FILE APPENDIX 6.1

```
 10 REM [.d000'!              assembly start
 20 REM ld hl,Basic'!         Basic address into hl
 30 REM ld a,(hl)'!           first character into a
 40 REM call .4646'!          statement handler
 50 REM ret'!                 return
 60 REM Basic'!               tokenised basic statement
 70 REM db .91'!              PRINT
 80 REM db .ff'db .9b'!       HEX$
 90 REM $('!                  (
100 REM db .ff'db .97'!       PEEK
110 REM $(&HF6B1)'!           (&HF6B1)
120 REM db .f1'!              +
130 REM $&H100'!              &H100
140 REM db .f3'!              *
150 REM db .ff'db .97'!       PEEK
160 REM $(&HF6B2)):'!         (&HF6B2))
170 REM ]
```

Assemble the file in the normal way and call the routine with Z$ = USR1(0). The current address of the stack pointer will be printed on the screen in hex.

Note that any ASCII characters in the routine must be in upper case. So the (&HF6B2) and other strings are all in upper case.

Using the statement handler can reduce the most complicated routines to simple proportions. Remember that there is 32K of powerful basic ROM in your MSX — using the built in routines can save vast amounts of space in your machine code programs.

APPENDIX 7

HOOK JUMPS

In the MSX computers there are many HOOK JUMPS provided so that the programmer can "HOOK" or attach his own machine code routine into a basic ROM routine. The hook jumps are situated in the systems area of memory and each hook consists of five bytes. Each of the bytes normally contains the number 201 which is the MC code for return.

At the start of many ROM routines there is a call to a hook which normally returns immediately. In order to use the hook you must place a jump to your own routine in the hook — this is illustrated by the following source file:

SOURCE FILE APPENDIX 7.1

```
 10 REM [.d000'!              start address
 20 REM ld a,.c3'!            code for jump
 30 REM ld (.ff43),a'!        put it in hook gone
 40 REM ld hl,Start'!         address for jump
 50 REM ld (.ff44),hl'!       put it in hook gone + 1
 60 REM ret'!                 return
 70 REM Start
 80 REM cp .m91'!             check for bracket
 90 REM ret nz'!              no bracket so ret
100 REM inc sp'!              remove rom return
110 REM inc sp'!              address from stack
120 REM inc hl'!              hl to next instruction
130 REM push hl'!             save it
140 REM ld hl,Str'!           point hl to Str
150 REM call .73e5'!          play it
160 REM pop hl'!              recover hl
170 REM ret'!                 back to basic
180 REM Str
190 REM $"t255cdef":'!        music string
200 REM ]'!                   end of source
```

This little program initialises the hook jump HOOK GONE so that the open square bracket character "[" becomes a command to play a music string. Assemble the file in the normal way then run it with Z = USR1(0). Now whenever you press [followed by ENTER the music string will play — this can be used in command or in program mode.

HOOK GONE is a very useful hook which is visited by all basic statements before syntax check. This means that you can define your own basic words — as we did in the given source file. To avoid problems please remember the following rules for HOOK GONE:

1) When the hook is called the "a" register contains the token of the current basic word. The first instruction in the Start routine is a check for our new word i.e. "[". If the current word is not a "[" then the program returns to the ROM — this is essential to maintain compatibility with all existing basic words.

2) When the ROM calls the hook, the return address on the stack is a return to the ROM. Normally when you hook in your own routine you want to return to your basic program, and not to the ROM, and so you must remove the ROM return address from the stack. This is done by incrementing the stack pointer twice thus leaving the basic return address at the top of the stack.

3) The address in the hl register is a pointer to the current position in the basic program — this address must be preserved so that the return to basic is correct. In our HOOK GONE routine the hl register is incremented so that hl points to the next basic instruction and not to the "[". After incrementing the hl register it is saved on the stack.

4) Finally after execution of the "hooked" routine the hl register is restored before returning to basic.

NOTE — Whenever you use any hook you must ascertain the condition of the STACK and the Z80 REGISTERS when the hook is called from the ROM. This knowledge is needed so that you can avoid a system crash or error condition on return to basic. Remember that the conditions could be different for each hook so you should disassemble the first section of the basic routine, which calls the hook, to obtain the necessary information.

Hook Gone is so useful you may never need any more hooks however for completeness a list of basic word hooks follows:

BASIC WORD	HOOK ADDRESS
DSKO$	&HFDEF
SET	&HFDF4
NAME	&HFDF9

BASIC WORD	HOOK ADDRESS
KILL	&HFDFE
IPL	&HFE03
COPY	&HFE08
CMD	&HFE0D
DSKF	&HFE12
DSKI$	&HFE17
ATTR$	&HFE1C
LSET	&HFE21
RSET	&HFE26
FIELD	&HFE2B
MKI$	&HFE30
MKS$	&HFE35
MKD$	&HFE3A
CVI	&HFE3F
CVS	&HFE44
CVD	&HFE49
MERGE	&HFE67
SAVE	&HFE6C
FILES	&HFE7B
LOC	&HFE99
LOF	&HFE9E
EOF	&HFEA3
FPOS	&HFEA8
WIDTH	&HFF84
LIST	&HFF89
SCREEN	&HFFC0
PLAY	&HFFC5

The majority of basic word hooks work as follows:

The basic word calls the basic ROM routine and the ROM routine calls the hook. When the hook is called there are two addresses on the stack namely the ROM return address and the BASIC return address. The stack looks like this:

```
        BASIC return address
TOP     ROM return address
```

The HL register pair contains the address of the character immediately after the basic word. If the basic instruction is PRINT "ABC" then the HL register pair would contain the address of the quotes (") at the start of "ABC".

To ensure a controlled return to basic it is essential that the address in HL is preserved.

Here is a simple use for the hook associated with the basic word LIST. When the hook is called the return address is to the ROM. If you remove this address from the stack then the return address is to the basic. This effectively disables the LIST command and prevents listing of the basic program. To remove the address from the stack we POP BC at the LIST HOOK. Use the following basic instruction to disable LIST:

POKE &HFF89,&HC1

OTHER HOOKS

Two hooks associated with the computer interrupts and timer are:

HKEYI &HFD9A
HTIMI &HFD9F

These hooks are called 50 — 60 times per second when the computer interrupts are not disabled. A possible software project for these hooks is a real time clock — NOTE that interrupts are always disabled during disc or tape I/O and so the clock would stop when loading or saving.

The last hook we are going to examine is:

HCHPU &HFDA4

This is the hook in the Character Put routine — here is a little program to use this hook to produce inverse characters:

SOURCE FILE APPENDIX 7.2

INVERSE CHARACTER GENERATOR

SETUP SECTION

```
10 REM [.f330'!           assembly start address
20 REM ld de,(.4)'!       address of CHR set
30 REM ld hl,.100'!       offset for space CHR
40 REM add hl,de'!        add to address
50 REM ex de,hl'!         put it in de
60 REM ld hl,.d00'!       address of space in VRAM
70 REM ld bc,.2f8'!       byte count
```

CREATE INVERSE CHARACTER SET

```
 80 REM Loop'!                  Loop label
 90 REM ld a,(de)'!             byte into a
100 REM cpl'!                   flip the bits
110 REM call .4d'!              send byte to VRAM
120 REM cpi'!                   inc hl & dec bc
130 REM inc de'!                increase RAM pointer
140 REM jp pe,Loop'!            if bc > 0 then Loop
```

INITIALISE HOOK

```
150 REM Init'!                  Initialise routine
160 REM ld a,.c3'!              jump instruction into a
170 REM ld (.fda4),a'!          put it into HCHPU
180 REM ld hl,Start'!           Start address into hl
190 REM ld (.fda5),hl'!         put it into HCHPU + 1
200 REM ld a,.0'!               inverse flag into a
210 REM ld (.fda7),a'!          put it into HCHPU + 3
220 REM ret'!                   return to basic
```

ROUTINE TO TOGGLE THE INVERSE FLAG

```
230 REM Start'!                 Start routine
240 REM cp .18'!                is CHR = SELECT
250 REM jr nz,Nott'!            no so goto Nott
260 REM ld a,(.fda7)'!          yes so get inverse flag in a
270 REM xor .1'!                change flag
280 REM ld (.fda7),a'!          put it back
290 REM ret'!                   return to CHPUT
```

ROUTINE TO CHECK AND INVERSE CHARACTER

```
300 REM Nott'!                  Nott toggle routine
310 REM cp .20'!                is CHR = SPACE
320 REM jr c,Tog'!              less than space so go Tog
330 REM ld a,(.fda7)'!          get inverse flag
340 REM and a'!                 set flag register
350 REM ret z'!                 no inverse so ret
360 REM pop bc'!                return address into bc
370 REM pop af'!                CHR code into a
380 REM set .7,a'!              convert to inverse
390 REM push af'!               put it back on stack
400 REM push bc'!               return address onto stack
410 REM ret'!                   return to CHPUT
```

ROUTINE TO SWITCH OFF INVERSE FLAG

```
420 REM Tog'!              subroutine Tog
430 REM ld a,.0'!          zero into a
440 REM ld (.fda7),a'!     reset inverse flag
450 REM ret']'!            return to CHPUT
```

Lets examine each section of this program source file:

SETUP SECTION

This section collects the address of the ROM character set from ROM addresses 4 and 5 into the DE register pair. This address is adjusted so that it points to the definition of the character SPACE. The HL register pair is set up to point to the VRAM address of CHR$(160) which is to become the first inverse character (ie. inverse space). The BC register is loaded with the number of bytes to be modified.

CREATE INVERSE CHARACTER SET

This section takes each ASCII character definition byte, changes binary 1's into 0's and 0's into 1's, and places the new definition into VRAM. NOTE the use of the cpi instruction — this two byte instruction increments HL, decrements BC, and sets the parity flag if BC is non zero (the parity flag is reset when BC is zero).

INITIALISE HOOK

This section sets up the HCHPU hook with a jump to our character inverse routine Start.

ROUTINE TO TOGGLE THE INVERSE FLAG

The address HCHPU + 3 is the inverse flag — this routine toggles the flag between 1 and 0 whenever the SELECT key is pressed.

ROUTINE TO CHECK AND INVERSE A CHARACTER

This routine checks if the current character is valid ASCII (ie. SPACE or greater), checks if inverse flag is set, inverses the character if flag is set and then returns to the ROM and puts the character on the screen. NOTE the stack and register conditions when the HCHPU is called:

STACK		BASIC RETURN ADDRESS
		AF REGISTER PAIR — A CONTAINS CHARACTER
	TOP	ROM RETURN ADDRESS
REGISTERS		HL — NEXT CHARACTER POINTER
		A — CHARACTER CODE

ROUTINE TO SWITCH OFF INVERSE FLAG

This routine switches off the inverse flag when the character has a code of less than 32. Each time you ENTER a line the inverse flag will therefore be switched off.

USING THE INVERSE PROGRAM

Load the source file and assemble in the usual way. Activate the program by typing:

```
DEFUSR3 = &HF330
Z = USR3(0)
```

Now when you want inverse characters you simply press the SELECT key and then type your characters. You can also select inverse by the program instruction PRINT CHR$(24) followed by the text to be printed.

SPECIAL NOTE

The inverse program operates in real time — this means that it is working away in the background all the time. The computer will therefore crash if you try to assemble again in the same memory space. You can abort the inverse program by typing POKE &HFDA4,&HC9.

FINAL NOTE ON HOOKS

MSX peripherals (eg. disc drive or RS232 card) usually make use of the hook jumps. It is therefore essential that you check the contents of a hook jump before you use it for your own purposes. The hook is unused if the hook address and the following 4 bytes all contain RET instructions ie. 201 decimal or C9 hex.

APPENDIX 8

READING INPUT DEVICES

The main input device is the keyboard and most of the keys produce an ASCII value which can be read in basic or in machine code. Several of the keys produce no ASCII values — these keys can only be detected by a direct read of the keyboard matrix. Use the following general code to detect a keypress of these special keys:

PROGRAM LIST APPENDIX 8.1

```
10 OUT &HAA,((INP(&HAA)AND240)ORY)
20 IF (INP(&HA9)ANDZ)<>0THEN10
```

Substitute values for Z and Y in order to select the desired key. This program will loop until the key, defined by Y and Z, is pressed.

1)	FUNCTION KEY 1:	Y = 6	Z = 32
2)	FUNCTION KEY 2:	Y = 6	Z = 64
3)	FUNCTION KEY 3:	Y = 6	Z = 128
4)	FUNCTION KEY 4:	Y = 7	Z = 1
5)	FUNCTION KEY 5:	Y = 7	Z = 2
6)	CTRL KEY:	Y = 6	Z = 2
7)	SHIFT KEY:	Y = 6	Z = 1
8)	GRAPH KEY:	Y = 6	Z = 4
9)	CODE KEY:	Y = 6	Z = 16
10)	CAPS LOCK:	Y = 6	Z = 8
11)	STOP KEY:	Y = 7	Z = 16

Note that shifted keys (and GRAPH/SHIFT or CODE/SHIFT combinations) produce the same values as unshifted ones — to detect between shifted and unshifted keys you should first read the shift key (and the code or graph keys) and then read the other key.

The full keyboard matrix is given below:

MSX KEYBOARD MATRIX (USA)

Y / Z	128	64	32	16	8	4	2	1
0	7	6	5	4	3	2	1	0
1	;	]	[	\	=	–	9	8
2	b	a	^	.	,		à	aá
3	j	i	h	g	f	e	d	c
4	r	q	p	o	n	m	l	k
5	z	y	x	w	v	u	t	s
6	F3	F2	F1	COD	CAP	GRP	CTR	SFT
7	ENT	SEL	BS	STP	TAB	ESC	F5	F4
8	RGT	DWN	CUP	LFT	DEL	INS	CLS	SPC
9					/	*	–	+

NOTES

a). The Y value is the row number and the Z value is the column number.

b) The bottom row of the matrix refers to the keypad.

c) CUP, LFT, DWN and RGT refer to the cursor direction keys.

d) To read the keyboard matrix from machine code proceed as follows:

   Load the A register with the Y value.
   Call 0141 hex.
   255 – Z is returned in the A register.
   255 is returned if no key in row Y has been pressed.

THE JOYSTICK

The joystick is another commonly used input device — the joystick direction can be read through PORT A of the PSG. The following mini-program illustrates the method of directly reading the joysticks:

PROGRAM LIST APPENDIX 8.2

```
10 S = 191                           ' joystick 1
20 OUT &HA0,&HF                      ' select PSG port B
30 OUT &HA1,(INP(&HA2)ANDS)          ' select joystick 1
40 OUT &HA0,&HE                      ' select PSG port A
50 Z = INP(&HA2)                     ' read PSG port A
60 PRINT BIN$(Z)                     ' print bits
70 GOTO 20                           ' do it again
```

NOTE: Use S = 255 for joystick 2.

The lower six bits of the number Z are significant — interpret as follows:

BIT 0 — If bit 0 = 0 then joystick is forward.

BIT 1 — If bit 1 = 0 then joystick is backward.

BIT 2 — If bit 2 = 0 then joystick is left.

BIT 3 — If bit 3 = 0 then joystick is right.

BIT 4 — If bit 4 = 0 then trigger A is pressed.

BIT 5 — If bit 5 = 0 then trigger B is pressed.

When reading the joystick position note that a 1 in any bit signifies no contact in the relevant direction. Note also that two directions are possible at one time on the same stick — so for example forward + left is equivalent to diagonally upwards to the left.

COMMENTS ON THE MAGIC OF MSX

Send to:

INTERSOFT (PTY) LTD., P.O. Box 5078, Johannesburg, 2000.

Name:..

Address:..

....

.... Code:....................

My constructive suggestions are:..

....

....

....

....

....

....

....

....

....

....

....

INTERSOFT gives its assurance that all information will be treated as strictly confidential (if applicable.)

INTERSOFT'S SPECIAL OFFER TO "THE MAGIC OF MSX" OWNERS

WORD WIZARD written by B.L. BURKE

WORD WIZARD is a word processing program for the MSX range of micro computers. The program provides extensive editing and printer control facilities.

All this for **R32.95** (incl. G.S.T.)
FREE! Postage and insurance within R.S.A.

Please send me WORD WIZARD.

Name:...

Address:..

....

.... Code:..............

Please debit my Visa Card or Master Card No.

Cheque ☐

Postal Orders ☐

Send the above coupon to: INTERSOFT (PTY) LTD., P.O. Box 5078, Johannesburg, 2000 by registered post and allow 21 days for delivery.