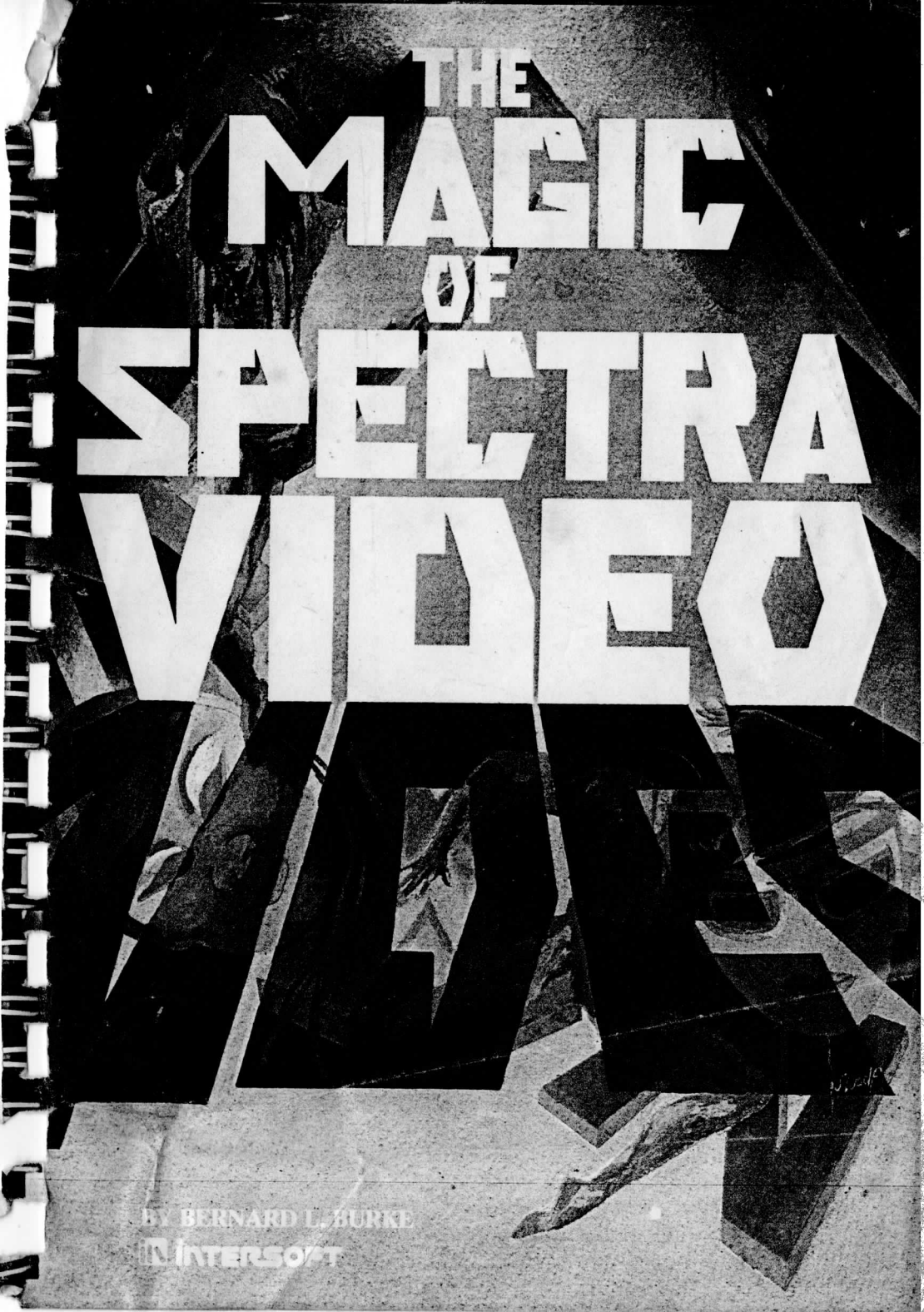

THE
MAGIC
OF
SPECTRA
VIDEO
BY BERNARD L. BURKE
INTERSOFT

COPY ONLY

Cover Design: Susan Woolf

Typesetting and Printing: Minit Print, Medical Centre.

INTERSOFT
P.O. Box 5078,
Johannesburg, 2000,
South Africa.
Tel: (011) 337-5806/7
Telex: 48-3868 SA.

THE MAGIC OF SPECTRAVIDEO

BY BERNARD L. BURKE

INTRODUCTION

This book is not a games book — there are plenty of those on the market already — THE MAGIC OF SPECTRAVIDEO 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 14. 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 15 and deals with machine code programming on the SPECTRAVIDEO. 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 a SPECTRAVIDEO USER for other SPECTRAVIDEO 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,
SOUTH AFRICA.

THANKS

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

Thanks to JAMES RALPH for distributing the MAGIC SPECTRAVIDEO. 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 "putting up" with me 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

TABLE OF CONTENTS

		Page
CHAPTER 1	**NUMBER SYSTEMS** The Decimal, Binary, Hexadecimal and Octal number systems are examined.	6
CHAPTER 2	**BITS BYTES AND OTHER WONDERFUL THINGS** Some microcomputer terms and concepts are discussed.	14
CHAPTER 3	**MEMORY MAPS AND SIGNPOSTS** The Spectravideo memory map is examined and a table of boundary addresses is provided. Program list 3.1 is a program which enables the user to switch into the second memory bank of the SVI 328.	18
CHAPTER 4	**BASIC PROGRAM AREA** This chapter looks at the basic program layout in memory. Several interesting program listings and a basic word/token table are provided.	26
CHAPTER 5	**VARIABLES AND ARRAYS** An examination of the way basic handles variables and arrays.	34
CHAPTER 6	**STRING SPACE** This chapter looks at strings and their location in memory. input/output files are also examined.	38
CHAPTER 7	**THE BASIC STACK** The operation of the stack is discussed and some program illustrations are given.	42

CHAPTER 8 **MACHINE SYSTEMS AREA** 45
The locations of the more useful system variables are given in table form and an autorun program for cload programs is presented.

CHAPTER 9 **THE VIDEO CHIP** 50
The TMS 9918A video chip is fully described including the various control registers and their contents.

CHAPTER 10 **DIRECT ACCESS TO THE VIDEO CHIP AND VIDEO RAM** 56
How to read from and write to the video chip registers and the video ram — essential information for the machine code programmer.

CHAPTER 11 **TEXT MODE** 59
The video chip in text mode — the character set and how to access up to 7 different character sets. The following character sets are presented:
The inverse set
The underline set
The upsidedown set

CHAPTER 12 **THE HIGH RESOLUTION SCREEN** 66
The screen 1 layout in detail including the 768 user defined graphics. Some interesting programs are presented.

CHAPTER 13 **VIDEO ENABLE/DISABLE** 75
Shows how to disable the screen to build a picture behind the scenes — enabling the screen then instantly displays your masterpiece.

CHAPTER 14 **THE VDP STATUS REGISTER** 77
Explains the use of the register to detect which sprites have collided.

CHAPTER 15 **MACHINE CODE** 79
The first chapter in the machine code section of the book — machine code and the Z80A chip are described.

CHAPTER 16 **THE SUPER ASSEMBLER** 88
A full Z80 assembler for your Spectravideo.

CHAPTER 17 **SUPER ASSEMBLER OPERATING INSTRUCTIONS** 95
Complete instructions for your new assembler.

CHAPTER 18 **SIMPLE SCREEN ROUTINES** 100
Some machine code routines for printing to the screen and getting characters from the keyboard.

CHAPTER 19 **MORE PRINTING ROUTINES** 105
Printing strings, screen formatting, as well as screen and cursor control commands.

CHAPTER 20 **THE SOUND OF MUSIC** 108
Music and sound routines from machine code — how to make continuous sound in your programs.

CHAPTER 21 **TRANSFERING VARIABLES FROM MACHINE CODE TO BASIC** 111
How to create basic variables from within your machine code routine.

CHAPTER 22 **SOME GRAPHICS ROUTINES** 114
A high resolution screen scroll routine and the machine code version of the sprite detection routine.

APPENDIX 1 **Z80 MACHINE CODE MNEMONICS** 120
A full list of all the Z80A instructions with brief explanations of each instruction group.

APPENDIX 2 **SPECTRAVIDEO ROM ROUTINES** 148
In which the formula for calculating the position of the basic word routines is presented.

APPENDIX 3 **INPUT/OUTPUT PORT TABLE** 149
Full list of the Z80 I/0 ports used by your SVI computer.

APPENDIX 4 **MORE ROM ROUTINES** 151
Some more of the routines which make your computer tick.

APPENDIX 5 **THE MAGIC OF SPECTRAVIDEO TAPE DIRECTORY** 153
A directory of the programs on the tape which accompanies this book.

APPENDIX 6 **THE BASIC STATEMENT HANDLER** 154
Shows how to use the Basic statement handler — this enables the user to execute any basic routine from within a machine code program.

APPENDIX 7 **HOOK JUMPS** 155
Shows how to hook your own routines into the rom. A full list of hook jumps is provided as well as a source file which creates a new basic word.

APPENDIX 8 **READING INPUT DEVICES** 158
Shows how to directly read the keyboard and the joystick.

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 SPECTRAVIDEO, in common with other computers, mostly uses the BINARY number system which counts up in 2's (DECIMAL counts in 10's). The SVI 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

Spectravideo 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

SVI 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

This system is not used often but the SPECTRAVIDEO computers use OCTAL in the disc system to keep track of the disc space allocated to various files.

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.

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 SVI 328 contains 917504 BITS and the SVI 318 contains 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 SVI 328 has 65536 bytes of continuous memory with addresses 0 to 65535. In the SVI 318 the address range is the same except that no memory is provided between addresses 32768 and 49151. Both 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 SPECTRAVIDEO — press ENTER

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

now type POKE 32773,130 — press ENTER

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

10 FOR SPECTRAVIDEO

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 SVI computer is as follows:

SVI 328 — from 32768 to 65535

SVI 318 — from 49148 to 65535

With both machines the area from 58624 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 SVI 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 PAGES (blocks) of 256 bytes each — so there are 4 pages to the KB, and 64 * 4 = 256 pages 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 Spectravideo 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 Spectravideo 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. Many of the routines in this book use machine code to increase the operating speed.

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 SVI 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 SVI 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 SVI is discussed later in the book.

CHAPTER 3

MEMORY MAPS AND SIGNPOSTS

The SVI 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).

Despite the limited addressing facility of the Z80A the SVI computers can be expanded to 160 KBYTES of user RAM in addition to the 16K of video RAM. This large memory is controlled by bank switching — banks of memory are switched IN and OUT as required so that the Z80A only sees a single 64K block at any one time.

INSTALLED MEMORY

Please refer to figures 3.0 — 3.3 whilst reading this section.

SVI computers have eight possible banks of memory — each bank can contain 32K Bytes. One lower bank (ie. BK01 or BK11 or BK21 or BK31) and one higher bank (ie. BK02 or BK12 or BK22 or BK32) are enabled at any one time.
SVI 318 — Contains 32K bytes of ROM (read only memory or "brains" which is located in bank BK01) and 32K bytes of RAM. 16K of the RAM is located in the upper half of bank BK02 and the other 16K is the video RAM.

SVI 328 — Contains 32K of ROM (as in the SVI 318) and 80K of RAM. 32K of the RAM is located in bank BK02, 32K is in bank BK21 and the remaining 16K is the video RAM.

MEMORY EXPANSION

a) The SVI 318 can be expanded to 160K user RAM by the addition of one 16K and two 64K expansion boards (one with BK21 and BK22 switched on and the other with BK31 and BK32 switched on).

b) The SVI 328 can be expanded to 160K of user RAM by the installation of two 64K expansion boards (one with BK22 and BK31 switched on and one with BK32 switched on — the other 32K in the second card cannot be used).

c) Note that memory installed in BK22 can be accessed in basic using the SWITCH command — this command exchanges the RAM banks BK02 and BK22 to give the user a second basic RAM.

d) BK21 is automatically switched in when the CP/M operating system is in use — this system is supplied with the disc drives. Program list 3.1 is a program to allow the user to utilise the memory in bank 21 for basic programming.

e) BK31 and BK32 are not accessible from basic and so are usually only of use to the machine code programmer.

BASIC RAM ORGANISATION

Figure 3.4 shows the layout of the BASIC RAM — BK02 or BK22. 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 in direct mode:

```
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 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.

PROGRAM LIST 3.1

BANK 21 SWITCH ROUTINE — MACHINE CODE BY BENNIE VAN DER MERWE

```
 10 CEAR200,&HF480
 20 FORX = &HF480T0&HF4F9
 30 READX$:POKEX,VAL("&H" + X$):NEXT
 40 DEFUSR = &HF480
 50 Z = USR(0):NEW
100 DATA 22,5E,FE,E5,3E,C3,32,57
110 DATA FF,21,B4,F4,22,58,FF,ED
120 DATA 73,5C,FE,F3,3E,0F,D3,88
130 DATA DB,90,32,64,FE,E6,FD,D3
140 DATA 8C,21,00,80,11,00,00,01
150 DATA 00,80,ED,B0,3A,64,FE,D3
160 DATA 8C,FB,E1,C9,FE,C9,C0,ED
170 DATA 73,5C,FE,DD,2A,03,FA,23
180 DATA 22,5E,FE,F3,3E,0F,D3,88
190 DATA DB,90,32,64,FE,E6,FD,D3
200 DATA 8C,21,00,00,11,00,80,4E
210 DATA EB,46,71,EB,70,23,13,7A
220 DATA B3,20,F4,3A,64,FE,D3,8C
230 DATA FB,ED,7B,5C,FE,CD,50,37
240 DATA DD,22,03,FA,2A,5E,FE,7E
250 DATA C1,C9
```

NOTE THAT THIS PROGRAM WILL ONLY WORK WITH THE SVI 328

Type in the program and CSAVE to tape. To execute type RUN followed by ENTER. The program will initialise BANK 21 as a second BASIC RAM storage area. The program will automatically delete after performing the initialisation so ensure that you CSAVE before running for the first time.

After running this program you can use the basic word SWITCH to switch between memory banks. One program can be stored in bank 21 and another in bank 02. The SWITCH command can be given in a program so programs can be run alternately in both banks.

EXAMPLE

RUN the SWITCH PROGRAM then type in the following program:

```
10 PRINT "THIS IS BANK 1"
20 SWITCH
30 GOTO 10
```

Type RUN followed by ENTER and the screen will display THIS IS BANK 1 and on the next line OK. Now type LIST followed by ENTER and notice that the program has dissappeared — You are now in the other bank. Type in the same program except change line 10 to PRINT "THIS IS BANK 2". Type RUN followed by ENTER and watch the computer continually switch and report current bank number.

FIGURE 3.0

SPECTRAVIDEO MEMORY BANK LAYOUT

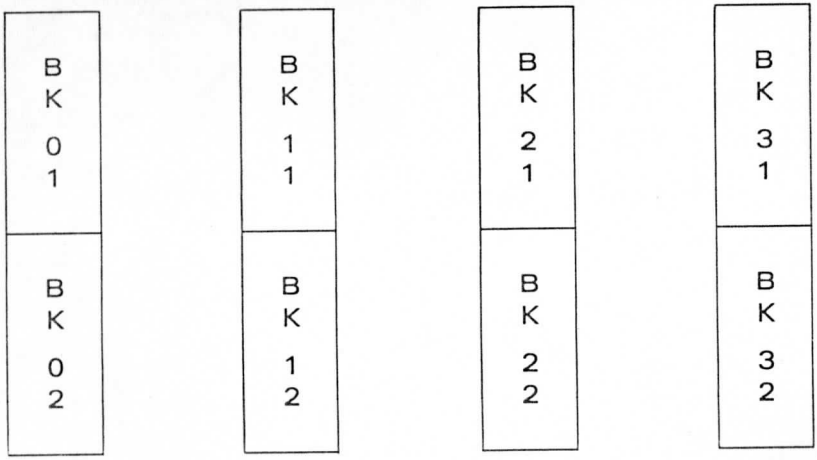
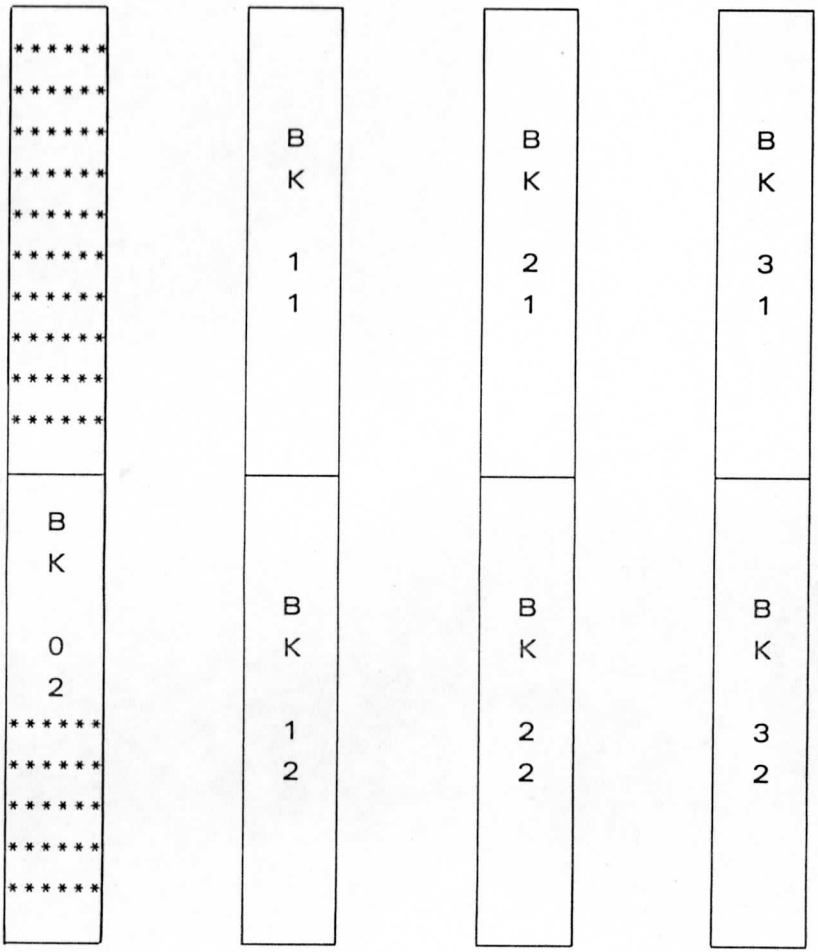


FIGURE 3.1

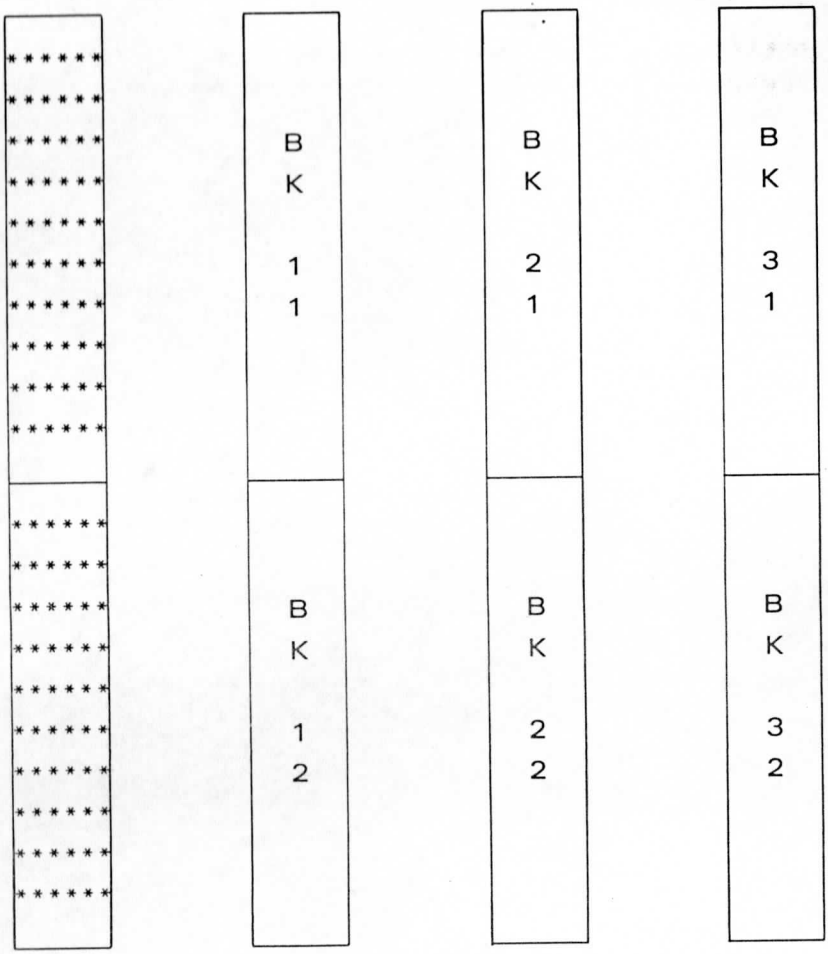
SPECTRAVIDEO 318 INSTALLED MEMORY



The *'s represent the installed memory.

FIGURE 3.2

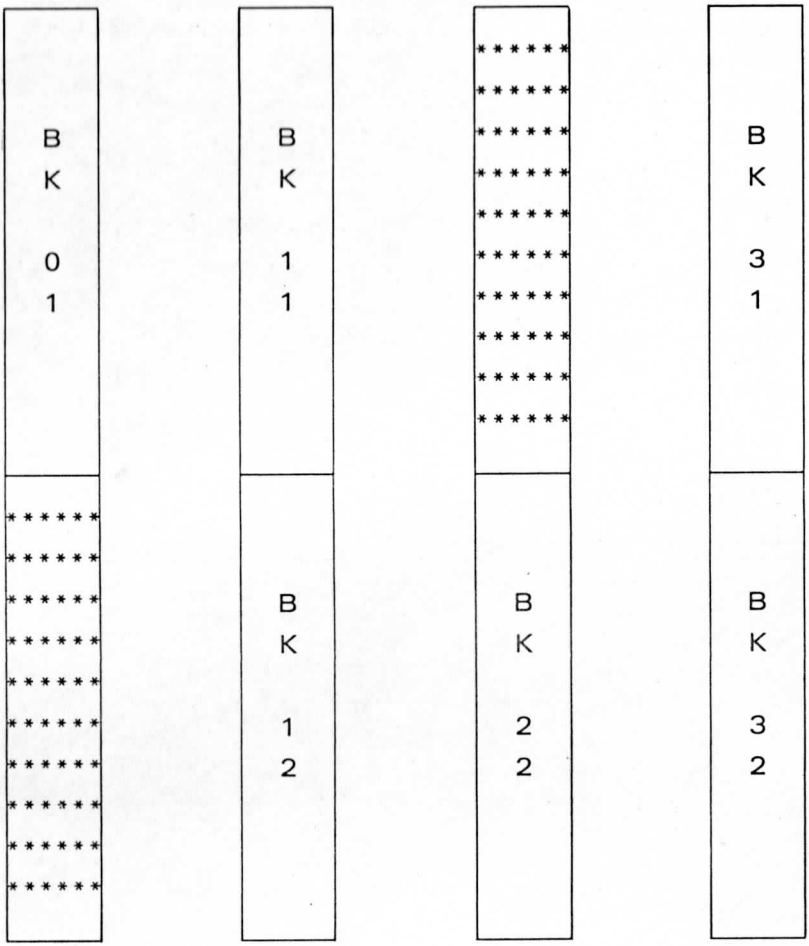
SPECTRAVIDEO 328 INSTALLED BASIC MEMORY



The *'s represent the installed memory.

FIGURE 3.3

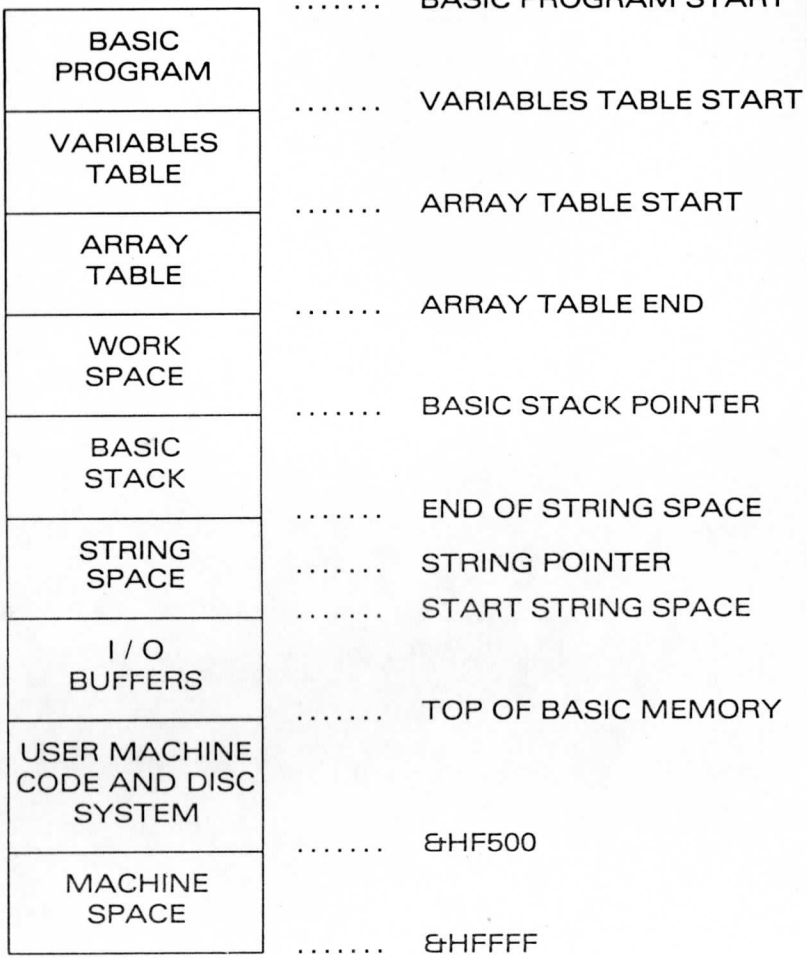
SPECTRAVIDEO 328 INSTALLED CP/M MEMORY



The *'s represent the installed memory.

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 SVI 328 this area starts at address &H8000 whilst the basic program area of the unexpanded SVI 318 starts at &HC000.

BASIC PROGRAM START ..... X = &HF54A Y = &HF54B

This system variable can be used within a program to decide if the computer is a 318 or a 328.

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 = &HF7EE Y = &HF7EF

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(&HF54A) + 256 * PEEK(&HF54B)
 60 X2 = PEEK(&HF7EE) + 256 * PEEK(&HF7EF)
 70 FORY = X1TOX2
 80 IFPEEK(Y) = 143ANDPEEK(Y + 1) = 90THENC = 1:NEXT
 90 IFC = 1ANDPEEK(Y) = 90THENPOKEY,127ELSEC = 0
100 NEXT
110 DELETE50—110
120 A$ = "EXCALIBER":REMZZZZZZZZZZZZZZZZZZZZ
130 CLS:INPUT"ENTER PASSWORD ";B$
140 IFB$ < >A$THENNEW:REMZZZZZZZZZZZZZZZZZZZZ
```

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(&HF54A)+256*PEEK(&HF54B)
 80 X2=PEEK(&HF7EE)+256*PEEK(&HF7EF)
 90 FORY=X1TOX2
100 IFPEEK(Y)=132ANDPEEK(Y+1)=34THENY=Y+1:
    GOTO110ELSENEXTY
110 FORZ=1TOLEN(A$):POKEY+Z,ASC(MID$(A$,Z,1)):
    NEXTZ
120 CLS:PRINT"PROGRAM PERSONALISATION COM-
    PLETE"
130 PRINT"PLEASE SAVE PROGRAM"
140 DELETE50-140
150 DATA"%%%%%%%%%%%%%%%%%%"
160 READN$: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 270. This variable always points to the next set of data in the basic program.

DATA POINTER ..... X = &HF7F4 Y = &HF7F5

PROGRAM LIST 4.3 — ANIMALS

```
 10 ONSTOPGOSUB210:STOPON
 20 SCREEN0,0:LOCATE,,0
 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 LOCATE0,14,1:INPUTAA$
 90 LOCATE,,0
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 SCREEN,1:LOCATE,,1: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 LOCATE0,17,1:INPUTAB$:LOCATE,,0
270 IFLEN(AB$)>19THENAB$=LEFT$(AB$,19)
280 DP=PEEK(&HF7F4)+256*PEEK(&HF7F5)
290 DP=DP-52
300 PE=PEEK(&HF7EE)+256*PEEK(&HF7EF)
310 FORPC=DPTOPE:IFPEEK(PC)=132ANDPEEK(PC+1)=
    64THEN320ELSENEXTPC
320 FORPK=1TOLEN(AA$)
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@@@@@@@@@@@@@@@@@@@@
```

NOTE THAT LINES 370 TO 760 ARE IDENTICAL AND CAN BE ENTERED BY EDITING THE LINE NUMBER ONLY.

BASIC KEYWORDS AND TOKENS

The MICROSOFT BASIC LANGUAGE of the SPECTRAVIDEO computers has 117 basic commands 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.

SV1 318/328 BASIC WORD AND TOKEN TABLE

AUTO	169	AND	248	ATTR$	233
BSAVE	206	BLOAD	205	BEEP	192
CLICK	200	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	DIAL	208
ELSE	161	END	129	ERASE	165
ERROR	166	ERL	225	ERR	226
EQV	251	FOR	130	FIELD	177
FILES	183	FN	222	GOTO	137
GO TO	137	GOSUB	141	GET	178
INPUT	133	IF	139	INSTR	229
IMP	252	INKEY$	236	IPL	213
KILL	212	KEY	199	LPRINT	157
LLIST	158	LET	136	LOCATE	216
LINE	175	LOAD	181	LSET	184
LIST	147	LFILES	187	MOTOR	204
MERGE	182	MOD	253	MON	203
MAX	202	MDM	207	NEXT	131
NAME	211	NEW	148	NOT	224
OPEN	176	OUT	156	ON	149
OR	249	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	238	SWITCH	201
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	239	USING	228
USR	221	VARPTR	231	VPOKE	198
WIDTH	160	WAIT	150	XOR	250
>	240	=	241	<	242
+	243	–	244	*	245
/	246	^	247		

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.

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 = &HF7EE Y = &HF7EF

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 = &HF7F0 Y = &HF7F1

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.
   iiii) 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 dimension 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 = &HF7F2 Y = &HF7F3

****

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 SVI 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 = &HF7A2 Y = &HF7A3

END STRING SPACE ..... X = &HF546 Y = &HF547

STRING POINTER ..... X = &HF7C7 Y = &HF7C8

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 = &HFDE6 Y = &HFDE7
START OF STRING SPACE ..... X = &HF7A2 Y = &HF7A3

Up to 16 files are available on your SVI 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 SVI 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.

FIELD COMMAND

You can use unopened files to format screen output by making use of FILES, FIELD, LSET and RSET commands.

```
eg.  10 WIDTH 40
     20 FIELD#1,40 AS A$
     30 FIELD#1,20 AS A1$,20 AS A2$
     40 B1$ = "LEFT SET"
     50 B2$ = "RIGHT SET"
     60 LSET A1$ = B1$:RSET A2$ = B2$
     70 PRINT A$
```

Line 20 defines the length of the file #1 field which we call A$. Line 30 divides this field into two equal parts and in line 60 we insert strings into the fields. Note that LSET inserts a string starting at the left side of the field and RSET inserts the string so that the last character will be at the right end of the field. The reader is urged to experiment with these commands and to enjoy the ease with which output can be formatted.

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. Note also that the usual tape screen prompts will operate as with normal program saving and loading.

****

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 = &HF7DD
 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 = &HF7DD
 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 = 62720 decimal or F500 hex.

SYSTEMS AREA END ADDRESS = 65535 decimal or FFFF hex.

TABLES

START ADDRESS	NAME	NO OF BYTES	DESCRIPTION
&HF52B	USR TABLE	20	10 * 2 BYTE USR ADDRESSES SET UP BY DEFUSR STATEMENT
&HF7F6	DEF TABLE	26	26 * 1 BYTE ENTRIES GIVING THE DEFAULT VARIABLE TYPE 2 = INTEGER 3 = STRING 4 = SINGLE 8 = DOUBLE CHANGE BY DEFINT ETC.
&HFA1E	FUNCTION STRING TABLE	160	10 * 16 BYTE ENTRIES ONE FOR EACH FUNCTION KEY CONTAINS CURRENT STRINGS
&HFB0E	MUSIC A	128	MUSIC QUEUE USED BY PLAY
&HFB8E	MUSIC B	128	MUSIC QUEUE USED BY PLAY

TABLES CONTINUED

START ADDRESS	NAME	NO OF BYTES	DESCRIPTION
&HFC0E	MUSIC C	128	MUSIC QUEUE USED BY PLAY
&HFCDA	VOICE A	36	STATIC DATA FOR MUSIC A
&HFCFF	VOICE B	36	STATIC DATA FOR MUSIC B
&HFD24	VOICE C	36	STATIC DATA FOR MUSIC C
&HFD69	FUNCTION FLAGS	10	INDICATES IF FUNCTION KEY TRAP IS ON = 1 OR OFF = 0
&HFDEB	TRAP TABLE	33	11 * 3 BYTE ENTRIES FOR F–KEY AND STOP TRAPS BYTE 1 OFF = 0 ON = 1 BYTES 2/3 ADDRESS OF TRAP GOSUB LINE
&HFE79	HOOK JUMP TABLE	315	105 * 3 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
&HF546	&HF547	END OF STRING SPACE
&HF54A	&HF54B	BASIC PROGRAM START
&HF7A2	&HF7A3	START OF STRING SPACE
&HF7C7	&HF7C8	STRING POINTER
&HF7CD	&HF7CE	POINTER TO END OF FOR LOOP
&HF7DB	&HF7DC	RESUME ADDRESS
&HF7DD	&HF7DE	STACK POINTER ADDRESS
&HF7DF	&HF7E0	LAST ERROR LINE NUMBER
&HF7E1	&HF7E2	CURRENT LINE USED BY LIST.
&HF7E5	&HF7E6	ERROR HANDLING LINE NUMBER
&HF7EA	&HF7EB	LAST LINE WHEN CTRL/STOP
&HF7EC	&HF7ED	RESTART ADDRESS USED BY CONT
&HF7EE	&HF7EF	START OF VARIABLES TABLE
&HF7F0	&HF7F1	START OF ARRAY TABLE
&HF7F2	&HF7F3	END OF ARRAY TABLE
&HF7F4	&HF7F5	ADDRESS OF NEXT DATA
&HF992	&HF993	ADDRESS OF FILE#0 BUFFER
&HFA17	&HFA18	ADDRESS OF QUEUE TABLES
&HFA1A	&HFA1B	END POINTER IN KEY BUFFER
&HFA1C	&HFA1D	START POINTER IN KEY BUFFER
&HFDE6	&HFDE7	TOP OF BASIC MEMORY
&HFE2E	&HFE2F	COUNTER FROM 0 TO 65535
&HFE30	&HFE31	CURRENT INTERVAL VALUE
&HFE32	&HFE33	INTERVAL COUNT DOWN

MORE SYSTEM VARIABLES AND FLAGS

ADDRESS	CONTENTS	POKE
&HF53F	LATEST ERROR NUMBER	NO
&HF543	SCREEN LINE LENGTH	YES
&HF7D6	AUTO LINE NUMBERING FLAG 1 = ON 0 = OFF	YES
&HF98C	NUMBER OF DISC DRIVES	NO
&HF98D	NUMBER OF FILES — MAXFILES	NO
&HFA02	CLICK SWITCH 1 = ON 0 = OFF	YES
&HFA03	CURSOR LINE	NO
&HFA04	CURSOR COLUMN	NO
&HFA05	CURSOR SWITCH 1 = ON 0 = OFF	YES
&HFA06	FUNCTION KEY DISPLAY SWITCH	NO
&HFA07	VDP REGISTER 1 CONTENTS	NO
&HFA0A	FOREGROUND COLOR	YES
&HFA0B	BACKGROUND COLOR	YES
&HFA0C	BORDER COLOR	YES

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

EVEN MORE SYSTEM VARIABLES AND FLAGS

ADDRESS	CONTENTS	POKE
&HFE35	INVERSE CHARACTERS FLAG 1 = ON 0 = OFF	YES
&HFE38	UPPER CASE CHARACTERS FLAG 1 = ON 0 = OFF	YES
&HFE3A	SCREEN MODE NUMBER	NO
&HFE3B	SPRITE SIZE 0 = 8 * 8 UNMAGNIFIED 1 = 8 * 8 MAGNIFIED 2 = 16 * 16 UNMAGNIFIED 3 = 16 * 16 MAGNIFIED	YES
&HFE3C	REGISTER 0 OF THE VDP	NO
&HFE3D	REGISTER 8 OF THE VDP	NO

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&HFD8D+X,Z:NEXT
20 POKE&HFA1C,&H8D:POKE&HFA1D,&HFD
30 POKE&HFA1A,&H91:POKE&HFA1B,&HFD
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.

CHAPTER 9

THE VIDEO CHIP

The SPECTRAVIDEO computers use the TMS 9918A video chip to handle all screen output. This chip has 4 different screen modes 3 of which are implemented on the SVI machines.

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

GENERAL

The SVI 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 the SVI 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 2 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 SVI 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 three display modes — screen 0, screen 1 and screen 2. 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 always set to zero on the SVI 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
GRAPHICS SCREEN 1	1	0	0
GRAPHICS SCREEN 2	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 the backdrop section 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 SPECTRAVIDEO communicates with the VDP and the VRAM through 4 INPUT/OUTPUT PORTS. The ports concerned are as follows:

1) OUTPUT PORT &H80 ..... WRITE VRAM DATA.

2) OUTPUT PORT &H81 ..... WRITE ADDRESS OR REGISTER NUMBER

3) INPUT PORT &H84 ..... READ VRAM DATA.

4) INPUT PORT &H85 ..... 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(&H85)
20 OUT&H81,X
30 OUT&H81,(YOR&H80)
```

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

NOTE

a) Z = INP(&H85) 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.

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 &H81.

6) Send the high byte out through port &H81.

7) Read the data in through port &H84.

PROGRAM LIST 10.1

VRAM DIRECT READ

```
 10 ' example VRAM direct read
 20 CLS
 30 VPOKE&H113,98
 40 VPOKE&H114,150
 50 X=INP(&H85)
 60 OUT&H81,&H13
 70 OUT&H81,&H1
 80 Z1=INP(&H84)
 90 Z2=INP(&H84)
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 &H81. The character codes are then read directly through port &H84 — 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 &H81.

6) Send the high byte or &H40 out through port &H81. 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 port &H80.

PROGRAM LIST 10.2

VRAM DIRECT WRITE

```
10 'example VRAM direct write
20 CLS
30 Z1 = 98
40 Z2 = 150
50 X = INP(&H85)
60 OUT&H81,&H13
70 OUT&H81,(&H1OR&H40)
80 OUT&H80,Z1
90 OUT&H80,Z2
```

This program illustrates the method of directly writing to the video RAM memory. Two VRAM codes are placed in variables Z1 and Z2. The VRAM destination address is output through port &H81 — first the low byte and then the high byte or &H40. The two data bytes are then output through port &H80. NOTE that the destination address automatically increments with each write operation.

CHAPTER 11

TEXT MODE

The SPECTRAVIDEO text mode is known as SCREEN 0 — this is the default mode which is always current when the computer is switched on.

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:	2312	00100000	32
	2313	01010000	80
	2314	10001000	136
	2315	10001000	136
	2316	11111000	248
	2317	10001000	136
	2318	10001000	136
	2319	00000000	0
CHARACTER S:	2456	01110000	112
	2457	10001000	136
	2458	10000000	128
	2459	01110000	112
	2460	00001000	8
	2461	10001000	136
	2462	01110000	112
	2463	00000000	0
CHARACTER E:	3112	00000111	7
(INVERSE)	3113	01111111	127
	3114	01111111	127
	3115	00001111	15
	3116	01111111	127
	3117	01111111	127
	3118	00000111	7
	3119	11111111	255

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.

PROGRAM LIST 11.1

```
10 ' character definitions
20 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 SVI 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.

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
100 ' call inverse set
110 X=INP(&H85)
120 OUT&H81,7
130 OUT&H81,(4OR&H80)
140 END
200 ' restore normal set
210 X=INP(&H85)
220 OUT&H81,1
230 OUT&H81,(4OR&H80)
```

****

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
100 ' call underline set
110 X=INP(&H85)
120 OUT&H81,2
130 OUT&H81,(4OR&H80)
140 END
200 ' restore normal set
210 X=INP(&H85)
220 OUT&H81,1
230 OUT&H81,(4OR&H80)
```

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
100 ' call upsidedown set
110 X=INP(&H85)
120 OUT&H81,3
130 OUT&H81,(4OR&H80)
140 END
200 ' restore normal set
210 X=INP(&H85)
220 OUT&H81,1
230 OUT&H81,(4OR&H80)
```

****

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 X = INP(&H85)
20 OUT &H81,&H1B
30 OUT &H81,(7OR&H80)
```

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

POKING AROUND ON THE SCREEN

Characters can be placed into any position on the screen using VPOKE and characters can be read from any position on the screen using VPEEK. It should be noted that the VRAM uses an adjusted set of codes for character display — these codes correspond to the position of the character definition within the pattern generator table (eg. space = 0, A = 33, etc.).

The VRAM codes are related to ASCII in the following way:

1) The 95 standard characters are numbered from 0 (representing space) to 94 (representing ~). These values are equal to ASCII values minus 32.

2) 95 represents the inverse (invisible) cursor.

3) The set of inverse characters is represented by numbers 96 to 190. Each inverse character code is equal to the standard character ASCII code plus 64.

4) Code 191 is the normal (visible) cursor. Put a few lines of text on the screen then — VPOKE 380,191 — which puts an image of the cursor in the center of the screen.

   SO WHAT? ..... Just an inverse space isn't it?

   Well no ..... It really is an image of the cursor ..... move the real cursor over those lines of text and watch the image reflect the characters over which the cursor passes.

5) The 56 graphics characters are represented by VRAM codes 192 to 247 (ie. SVI CODE plus 32).

6) When writing to the screen in the normal way with PRINT then the computer adjusts the codes to print the correct characters. When writing to the screen with VPOKE then the user must adjust the codes before poking them onto the screen.

****

That concludes the examination of the TEXT MODE — in the next chapter we look at the VDP in high resolution graphics mode.

Name Table
$1800 - $

Graphics II

0	$17FF	Pattern
$1800	$1aff	Name Table
$1600	$1FFF	Sprite Attr
$2000	$37FF	Color Table
$3800		sprite Pattern

CHAPTER 12

THE HIGH RESOLUTION SCREEN

The SPECTRAVIDEO high resolution screen (SCREEN 1) 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 GLOBAL COLORS
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 1 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 immediately become visible on the screen.

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

```
10 SCREEN 1
20 LOCATE 1,0
30 PRINT "A"
40 GOTO 40
```

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 SCREEN 1
20 LOCATE 1,0
30 PRINT "A"
40 LOCATE 9,0
50 PRINT "B"
60 GOTO 60
```

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 SCREEN 1
 20 LOCATE 1,0
 30 PRINT "A"
 40 LOCATE 9,0
 50 PRINT "B"
 60 FOR X = 0 TO 7
 70 VPOKE &H2000+X,&H6A
 80 VPOKE &H2008+X,&HA6
 90 NEXT
100 GOTO 100
```

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 SCREEN 1
 20 LOCATE 1,0
 30 PRINT "A"
 40 LOCATE 9,0
 50 PRINT "B"
 60 FOR X = 0 TO 7
 70 VPOKE &H2000+X,&H6A
 80 VPOKE &H2008+X,&HA6
 90 NEXT
100 FOR X = 0 TO 255
110 VPOKE &H1800+X,0
120 NEXT
130 FOR X = 0 TO 255
140 VPOKE &H1800+X,1
150 NEXT
160 FOR X = 0 TO 255
170 VPOKE &H1800+X,2
180 NEXT
190 VPOKE &H1850,0
200 VPOKE &H18FF,1
210 GOTO 210
```

NOTES

a) Lines 100 to 120 fill the top section of the screen with user defined graphic 0 — the "A".

b) Lines 130 to 150 fill the top section of the screen with user defined graphic 1 — the "B".

c) Lines 160 to 180 fill the top section of the screen with user defined graphic 2 — undefined and therefore just blank.

d) Finally lines 190 and 200 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 1 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 1 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 1 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.

PROGRAM LIST 12.1

To conclude this chapter the program list 12.1 is presented — this program is used to mix your own titles with the SPECTRAVIDEO logo on screen 1. The machine code reads the logo off the screen into normal memory and then puts the logo back on the screen with your own titles.

```
 10 REM SVI logo mix — SVI 328 only
 20 CLEAR200,50000!
 30 DEFINTZ
 40 FORZ=1TO110
 50 READX$
 60 POKE 50000!+Z,VAL("&H"+X$)
 70 NEXT
 80 DEFUSR0=50001!
 90 DEFUSR1=50055!
100 LOCATE,,0
110 DEFUSR2=&H4782
120 SCREEN1
130 Z=USR2(0)
140 Z=USR0(0)
150 SCREEN0
160 SCREEN1
170 Z=USR1(0)
180 LOCATE60,40
190 COLOR11
200 PRINT"BERNARD L BURKE PRESENTS"
210 COLOR15
220 GOTO220
230 REM GET mid section screen
240 DATADB,85,F3,3E,00,D3,81,3E
250 DATA08,D3,81,21,C8,C3,01,00
260 DATA08,DB,84,77,23,0B,3E,00
270 DATAB8,20,F6,B9,20,F3,D3,81
280 DATA3E,28,D3,81,01,00,08,DB
290 DATA84,77,23,0B,3E,00,B8,20
300 DATAF6,B9,20,F3,FB,C9
310 REM PUT mid section screen
320 DATADB,85,F3,3E,00,D3,81,3E
330 DATA08,F6,40,D3,81,21,C8,C3
340 DATA01,00,08,7E,D3,80,23,0B
350 DATA3E,00,B8,20,F6,B9,20,F3
360 DATAD3,81,3E,68,D3,81,01,00
370 DATA08,7E,D3,80,23,0B,3E,00
380 DATAB8,20,F6,B9,20,F3,FB,C9
```

In the next chapter we look at the video disable facility in the VDP register 1.

CHAPTER 13

VIDEO ENABLE/DISABLE

By using the VIDEO ENABLE/DISABLE bit of VDP register 1 it is possible to make complicated graphics appear instantly on 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) from the register save address &HFA07.

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) Clear the VDP status register.

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

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

6) Restore the old value in the VDP register 1. NOTE that in this case we do not clear the status register because the VDP interrupt is disabled. The picture on the screen is instantly displayed.

This procedure is illustrated in program list 13.1.

PROGRAM LIST 13.1

```
10 COLOR15,4,4
20 SCREEN1
30 '
40 ' x=vdp reg1 data
50 '
60 X=PEEK(&HFA07)
70 '
80 ' disable screen
90 '
```

PROGRAM LIST 13.1 (CONTINUED)

```
100 Z = INP(&H85)
110 OUT&H81,(XAND&H9F)
120 OUT&H81,(1OR&H80)
130 CIRCLE(90,90),20,8
140 PAINT(90,90),8
150 LINE(10,10)—(180,50),11,BF
160 '
170 ' enable screen
180 '
190 OUT&H81,X
200 OUT&H81,(1OR&H80)
210 GOTO210
```

Change lines 120 to 140 in order to draw your own picture behind the scenes.

****

In the next chapter we conclude the discussion on the VDP with an examination of the use of the 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 1
 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=PEEK(&HFE3D)
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 remaining 5 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 190.
2) The routine at line 190 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 DEFINTA-Z
 20 SCREEN1
 30 ONSPRITEGOSUB190
 40 FORX=0TO7
 50 A$=A$+CHR$(255)
 60 NEXT
 70 DEFFNSC(S)=(PEEK(S)AND&B00100000)
 80 S=&HFE3D
 90 SPRITE$(0)=A$
100 FORP=1TO15
110 PUTSPRITEP,(50+P*10,P*10),P,0
120 NEXT
130 PD=(INT((RND(-TIME)*150)/10))*10
140 SPRITEON
150 FORZ=-20TO255
160 PUTSPRITE0,(Z,PD),3,0
170 NEXT
180 GOTO130
190 SPRITEOFF
200 FORX=4TO60STEP4
210 YP=VPEEK(&H1B00+X)
220 IFYP=209THENNEXTTELSEVPOKE&H1B00+X,209
230 FORWT=1TO50:NEXT
240 IFFNSC(S)=0THENPRINTX/4:RETURN130
250 VPOKE&H1B00+X,YP
260 NEXTX
270 RETURN130
```

CHAPTER 15

MACHINE CODE

The remainder of this book is devoted to an introduction to Z80 MACHINE CODE and its implementation on the SPECTRAVIDEO. Programs presented include a full Z80 assembler which the reader can key into his computer — NOTE that the assembler will be found 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 Spectravideo 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 "SPECTRAVIDEO" — 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 SPECTRAVIDEO

When the SVI 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 PROGRAM LIST 3.1 — in line 10 the CLEAR command is used to lower the top of memory to &HF480 before installing the 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 — see lines 40 and 50 in PROGRAM LIST 3.1.

Notice also in program 3.1 that 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 simple 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 is presented in the next chapter and your next task is to type in the assembler — or find it on the tape provided.

****

CHAPTER 16

THE SUPER ASSEMBLER

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

The given listing includes all the machine code and a loader program. Type in the listing and CSAVE to tape — now place a new tape in the data cassette and RUN the program. The machine code assembler will load into memory and then BSAVE to tape.

For future runs of the assembler you may CLOAD and RUN the basic listing or you may BLOAD the machine code version directly. If you directly BLOAD the MC version please remember to reserve space first:

type CLEAR 200,48000 followed by ENTER before BLOAD.

NOTES

1) The given version of the assembler is for the SVI 328 or the expanded SVI 318 — the version is suitable for disc or tape.

2) The SUPER ASSEMBLER is available on the accompanying tape. The tape contains two versions of the assembler.

   a) SA328 — suitable for the SVI 328 or the expanded 318.

   b) SA318 — suitable for the unexpanded SVI 318.

3) When you type in the listing take great care to check each data line for accuracy — the assembler will not work properly if any of the data is entered incorrectly.

4) Entering the listing will take some time so please CSAVE the growing listing frequently — there is nothing worse than coming near to the end of a large listing and losing the lot due to a power failure or some such mishap.

PROGRAM LISTING 16.1

SUPER ASSEMBLER

```
 10 ' SUPER ASSEMBLER
 20 '
 30 '
 40 '
 50 '
 60 '
 70 '
 80 CLEAR100,47999!
 90 FORX=48000!TO52400!:POKEX,0:NEXT
100 FORX=48000!TO49407!:READX$:POKEX,VAL("&H"+
    X$):NEXT
110 FORX=49984!TO52400!:READX$:POKEX,VAL("&H"+
    X$):NEXT
120 BSAVE"SA328",48000!,52400!
130 END
140 DATA 69,6E,20,62,2C,28,63,29,00,6F,75,74,20,28,63,29
150 DATA 2C,62,00,73,62,63,20,68,6C,2C,62,63,00,6C,64,20
160 DATA 28,23,29,2C,62,63,00,6E,65,67,00,72,65,74,6E,00
170 DATA 20,20,00,6C,64,20,69,2C,61,00,69,6E,20,63,2C,28
180 DATA 63,29,00,6F,75,74,20,28,63,29,2C,63,00,61,64,63
190 DATA 20,68,6C,2C,62,63,00,6C,64,20,62,63,2C,28,23,29
200 DATA 00,20,20,00,72,65,74,69,00,20,20,00,6C,64,20,72
210 DATA 2C,61,00,69,6E,20,64,2C,28,63,29,00,6F,75,74,20
220 DATA 28,63,29,2C,64,00,73,62,63,20,68,6C,2C,64,65,00
230 DATA 6C,64,20,28,23,29,2C,64,65,00,20,20,00,20,20,00
240 DATA 20,20,00,6C,64,20,61,2C,69,00,69,6E,20,65,2C,28
250 DATA 63,29,00,6F,75,74,20,28,63,29,2C,65,00,61,64,63
260 DATA 20,68,6C,2C,64,65,00,6C,64,20,64,65,2C,28,23,29
270 DATA 00,20,20,00,20,20,00,20,20,00,6C,64,20,61,2C,72
280 DATA 00,69,6E,20,68,2C,28,63,29,00,6F,75,74,20,28,63
290 DATA 29,2C,68,00,73,62,63,20,68,6C,2C,68,6C,00,20,20
300 DATA 00,20,20,00,20,20,00,20,20,00,72,72,64,00,69,6E
310 DATA 20,6C,2C,28,63,29,00,6F,75,74,20,28,63,29,2C,6C
320 DATA 00,61,64,63,20,68,6C,2C,68,6C,00,20,20,00,20,20
330 DATA 20,00,20,20,20,00,20,20,00,72,6C,64,00,69,6E,20
340 DATA 66,2C,28,63,29,00,20,20,00,73,62,63,20,68,6C,2C
350 DATA 73,70,00,6C,64,20,28,23,29,2C,73,70,00,20,20,00
360 DATA 20,20,00,20,20,00,20,20,00,69,6E,20,61,2C,28,63
370 DATA 29,00,6F,75,74,20,28,63,29,2C,61,00,61,64,63,20
380 DATA 68,6C,2C,73,70,00,6C,64,20,73,70,2C,28,23,29,00
```

```
390 DATA 00,00,6C,64,69,00,63,70,69,00,69,6E,69,00,6F,75
400 DATA 74,69,00,20,20,00,20,20,00,20,20,00,20,20,00,6C
410 DATA 64,64,00,63,70,64,00,69,6E,64,00,6F,75,74,64,00
420 DATA 20,20,00,20,20,00,20,20,00,20,20,00,6C,64,69,72
430 DATA 00,63,70,69,72,00,69,6E,69,72,00,6F,74,69,72,00
440 DATA 20,20,00,20,20,00,20,20,00,20,20,00,6C,64,64,72
450 DATA 00,63,70,64,72,00,69,6E,64,72,00,6F,74,64,72,00
460 DATA 00,00,6E,6F,70,00,6C,64,20,62,63,2C,23,00,6C,64
470 DATA 20,28,62,63,29,2C,61,00,69,6E,63,20,62,63,00,69
480 DATA 6E,63,20,62,00,64,65,63,20,62,00,6C,64,20,62,2C
490 DATA 23,00,72,6C,63,61,00,65,78,20,61,66,2C,61,66,22
500 DATA 00,61,64,64,20,68,6C,2C,62,63,00,6C,64,20,61,2C
510 DATA 28,62,63,29,00,64,65,63,20,62,63,00,69,6E,63,20
520 DATA 63,00,64,65,63,20,63,00,6C,64,20,63,2C,23,00,72
530 DATA 72,63,61,00,64,6A,6E,7A,2C,23,00,6C,64,20,64,65
540 DATA 2C,23,00,6C,64,20,28,64,65,29,2C,61,00,69,6E,63
550 DATA 20,64,65,00,69,6E,63,20,64,00,64,65,63,20,64,00
560 DATA 6C,64,20,64,2C,23,00,72,6C,61,00,6A,72,20,23,00
570 DATA 61,64,64,20,68,6C,2C,64,65,00,6C,64,20,61,2C,28
580 DATA 64,65,29,00,64,65,63,20,64,65,00,69,6E,63,20,65
590 DATA 00,64,65,63,20,65,00,6C,64,20,65,2C,23,00,72,72
600 DATA 61,00,6A,72,20,6E,7A,2C,23,00,6C,64,20,68,6C,2C
610 DATA 23,00,6C,64,20,28,23,29,2C,68,6C,00,69,6E,63,20
620 DATA 68,6C,00,69,6E,63,20,68,00,64,65,63,20,68,00,6C
630 DATA 64,20,68,2C,23,00,64,61,61,00,6A,72,20,7A,2C,23
640 DATA 00,61,64,64,20,68,6C,2C,68,6C,00,6C,64,20,68,6C
650 DATA 2C,28,23,29,00,64,65,63,20,68,6C,00,69,6E,63,20
660 DATA 6C,00,64,65,63,20,6C,00,6C,64,20,6C,2C,23,00,63
670 DATA 70,6C,00,6A,72,20,6E,63,2C,23,00,6C,64,20,73,70
680 DATA 2C,23,00,6C,64,20,28,23,29,2C,61,00,69,6E,63,20
690 DATA 73,70,00,69,6E,63,20,28,68,6C,29,00,64,65,63,20
700 DATA 28,68,6C,29,00,6C,64,20,28,68,6C,29,2C,23,00,73
710 DATA 63,66,00,6A,72,20,63,2C,23,00,61,64,64,20,68,6C
720 DATA 2C,73,70,00,6C,64,20,61,2C,28,23,29,00,64,65,63
730 DATA 20,73,70,00,69,6E,63,20,61,00,64,65,63,20,61,00
740 DATA 6C,64,20,61,2C,23,00,63,63,66,00,00,00,72,65,74
750 DATA 20,6E,7A,00,70,6F,70,20,62,63,00,6A,70,20,6E,7A
760 DATA 2C,23,00,6A,70,20,23,00,63,61,6C,6C,20,6E,7A,2C
770 DATA 23,00,70,75,73,68,20,62,63,00,20,20,00,20,20,00
780 DATA 72,65,74,20,7A,00,72,65,74,00,6A,70,20,7A,2C,23
790 DATA 00,20,20,00,63,61,6C,6C,20,7A,2C,23,00,63,61,6C
800 DATA 6C,20,23,00,20,20,00,20,20,00,72,65,74,20,6E,63
810 DATA 00,70,6F,70,20,64,65,00,6A,70,20,6E,63,2C,23,00
820 DATA 6F,75,74,20,28,23,29,2C,61,00,63,61,6C,6C,20,6E
```

```
830 DATA 63,2C,23,00,70,75,73,68,20,64,65,00,20,20,00,20
840 DATA 20,00,72,65,74,20,63,00,65,78,78,00,6A,70,20,63
850 DATA 2C,23,00,69,6E,20,61,2C,28,23,29,00,63,61,6C,6C
860 DATA 20,63,2C,23,00,20,20,00,20,20,00,20,20,00,72,65
870 DATA 74,20,70,6F,00,70,6F,70,20,68,6C,00,6A,70,20,70
880 DATA 6F,2C,23,00,65,78,20,28,73,70,29,2C,68,6C,00,63
890 DATA 61,6C,6C,20,70,6F,2C,23,00,70,75,73,68,20,68,6C
900 DATA 00,20,20,00,20,20,00,72,65,74,20,70,65,00,6A,70
910 DATA 20,28,68,6C,29,00,6A,70,20,70,65,2C,23,00,65,78
920 DATA 20,64,65,2C,68,6C,00,63,61,6C,6C,20,70,65,2C,23
930 DATA 00,20,20,00,20,20,00,20,20,00,72,65,74,20,70,00
940 DATA 70,6F,70,20,61,66,00,6A,70,20,70,2C,23,00,64,69
950 DATA 00,63,61,6C,6C,20,70,2C,23,00,70,75,73,68,20,61
960 DATA 66,00,20,20,00,20,20,00,72,65,74,20,6D,00,6C,64
970 DATA 20,73,70,2C,68,6C,00,6A,70,20,6D,2C,23,00,65,69
980 DATA 00,63,61,6C,6C,20,6D,2C,23,00,00,00,01,01,00,00
990 DATA 00,00,00,00,00,00,01,00,00,00,5D,00,00,00,00,00
1000 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,15
1010 DATA 80,00,00,50,F4,02,00,00,00,00,00,00,00,00,00,00
1020 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,52
1030 DATA 21,CC,C0,01,83,02,AF,77,23,0B,78,B1,20,F8,3A,02
1040 DATA FA,32,CC,C0,AF,32,02,FA,21,C2,C3,CD,9A,45,3E,01
1050 DATA 32,CD,C0,21,07,80,22,EF,C0,21,80,C3,CD,7D,69,C9
1060 DATA 53,75,70,65,72,20,41,73,73,65,6D,62,6C,65,72,20
1070 DATA 56,65,72,73,69,6F,6E,20,36,2E,30,0D,0A,28,43,29
1080 DATA 20,31,39,38,35,20,62,79,20,42,65,6E,6E,69,65,20
1090 DATA 76,61,6E,20,64,65,72,20,4D,65,72,77,65,2E,0D,0A
1100 DATA 0A,00,30,00,C7,C8,C9,CA,3E,68,32,0E,C4,3E,26,32
1110 DATA 0D,C4,2B,23,7E,FE,2E,28,FA,FE,6F,20,04,32,0E,C4
1120 DATA 23,FE,6E,20,06,3E,62,32,0E,C4,23,FE,6D,CA,21,C4
1130 DATA 11,0E,C4,13,7E,FE,00,28,08,FE,29,28,04,12,23,18
1140 DATA F2,AF,12,21,0D,C4,CD,CA,14,2A,25,F9,C9,26,68,66
1150 DATA 34,35,30,00,00,2E,66,66,00,00,00,00,00,00,00,00
1160 DATA 00,E5,D1,21,00,00,13,1A,FE,29,28,15,FE,00,28,11
1170 DATA D5,06,09,E5,D1,19,10,FD,16,00,D6,30,5F,19,D1,18
1180 DATA E5,22,25,F9,C9,49,4A,4B,4C,4D,3E,23,77,23,E5,D1
1190 DATA 7E,FE,00,28,0B,FE,2C,28,07,FE,29,28,03,23,18,F0
1200 DATA 7E,12,23,13,FE,00,20,F8,12,C9,6E,6F,70,71,72,73
1210 DATA 74,75,22,39,C2,21,42,C1,22,3B,C2,0E,01,3A,CD,C0
1220 DATA 47,2A,F3,C0,22,D4,C0,3A,F5,C0,FE,00,C8,2A,3B,C2
1230 DATA 11,06,00,19,22,3B,C2,E5,D1,2A,39,C2,1A,FE,00,20
1240 DATA 08,7E,FE,00,20,03,C3,CE,C4,1A,BE,20,04,13,23,18
1250 DATA EB,1A,FE,00,20,08,7E,FE,29,20,03,C3,CE,C4,0C,05
1260 DATA C2,8D,C4,21,E0,C4,CD,7D,69,1E,11,C3,07,09,21,F6
```

```
1270 DATA C0,23,23,0D,20,FB,7E,32,D4,C0,23,7E,32,D5,C0,C9
1280 DATA 0A,55,6E,64,65,66,69,6E,65,64,20,6C,61,62,6C,65
1290 DATA 2E,0A,0D,00,F8,F9,FA,FB,FC,FD,FE,00,01,02,21,DA
1300 DATA C0,06,13,AF,77,23,10,FB,3A,84,FD,FE,FE,28,F9,3A
1310 DATA 86,FD,FE,EF,20,17,21,21,C5,CD,7D,69,1E,11,C3,07
1320 DATA 09,0A,45,73,63,61,70,65,2E,0D,0A,0A,00,2A,EF,C0
1330 DATA 7E,FE,27,20,01,23,FE,00,20,04,11,07,00,19,7E,FE
1340 DATA 24,CA,86,C5,FE,21,CA,65,C5,11,DA,C0,0E,00,7E,FE
1350 DATA 00,28,0A,FE,27,28,06,0C,12,23,13,18,F1,22,EF,C0
1360 DATA 79,32,D6,C0,C9,3E,0A,DF,7E,DF,23,7E,FE,27,28,07
1370 DATA FE,00,28,03,DF,18,F3,3E,0A,DF,3E,0A,DF,3E,0D,DF
1380 DATA 22,EF,C0,C3,FE,C4,DF,ED,5B,F3,C0,1B,23,13,7E,FE
1390 DATA 27,28,0A,FE,00,28,06,12,D5,DF,D1,18,EF,ED,53,F3
1400 DATA C0,18,D7,A8,A9,AA,AB,AC,AD,AE,AF,B0,B1,B2,B3,B4
1410 DATA B5,B6,3A,DA,C0,21,DB,C0,FE,5B,20,09,CD,C8,C3,22
1420 DATA F3,C0,C3,5D,C6,FE,5D,C2,20,C6,3A,F5,C0,3C,32,F5
1430 DATA C0,FE,02,28,11,CD,78,37,21,80,C3,CD,7D,69,21,07
1440 DATA 80,22,EF,C0,AF,C9,3E,0A,DF,21,0B,C6,CD,7D,69,2A
1450 DATA F3,C0,CD,3D,CC,3E,0D,DF,3E,0A,DF,3E,0A,DF,3A,CC
1460 DATA C0,32,02,FA,3E,02,32,93,F7,E1,C9,46,69,72,73,74
1470 DATA 20,63,6C,65,61,72,20,61,64,64,72,65,73,73,3A,00
1480 DATA FE,64,C2,59,C6,7E,FE,62,20,11,23,23,CD,C8,C3,7D
1490 DATA 2A,F3,C0,77,23,22,F3,C0,C3,60,C6,FE,77,20,1A,23
1500 DATA 23,CD,C8,C3,DD,2A,F3,C0,DD,75,00,DD,74,01,DD,23
1510 DATA DD,23,DD,22,F3,C0,C3,60,C6,3E,FF,C9,C9,3E,0A,DF
1520 DATA 21,DA,C0,CD,7D,69,3E,0D,DF,3E,0A,DF,AF,C9,74,75
1530 DATA 3A,DA,C0,FE,5B,30,06,FE,41,38,02,18,03,3E,FF,C9
1540 DATA 3A,D6,C0,FE,06,D2,03,C7,3A,F5,C0,FE,01,CA,CF,C6
1550 DATA 3A,CD,C0,FE,28,CA,E3,C6,3A,CD,C0,21,F6,C0,23,23
1560 DATA 3D,20,FB,3A,F3,C0,77,23,3A,F4,C0,77,3A,CD,C0,21
1570 DATA 42,C1,11,06,00,19,3D,20,FC,EB,21,DA,C0,06,06,7E
1580 DATA 12,23,13,10,FA,3A,CD,C0,3C,32,CD,C0,C3,CF,C6,3E
1590 DATA 0A,DF,21,DA,C0,CD,7D,69,3E,0D,DF,3E,0A,DF,3E,0A
1600 DATA DF,AF,C9,21,EE,C6,CD,7D,69,1E,11,C3,07,09,0A,54
1610 DATA 6F,6F,20,6D,61,6E,79,20,6C,61,62,6C,65,73,2E,0D
1620 DATA 0A,0A,00,21,0E,C7,CD,7D,69,1E,11,C3,07,09,0A,4C
1630 DATA 61,62,6C,65,20,74,6F,6F,20,6C,6F,6E,67,2E,0D,0A
1640 DATA 0A,00,29,2A,2B,2C,2D,2E,2F,30,31,32,33,34,35,36
1650 DATA 37,38,39,3A,3B,3C,3D,3E,AF,32,D1,C0,3A,DA,C0,FE
1660 DATA 64,20,0D,3A,DB,C0,FE,6A,20,06,21,DF,C0,C3,76,C7
1670 DATA 3A,DA,C0,FE,6A,20,07,3A,DB,C0,FE,72,28,03,3E,FF
1680 DATA C9,21,E0,C0,3A,DF,C0,FE,2C,28,0B,2B,3A,DE,C0,FE
1690 DATA 2C,28,03,21,DD,C0,3E,FF,32,D1,C0,7E,FE,2D,C2,8E
1700 DATA C7,E5,23,CD,C8,C3,AF,95,32,D2,C0,C3,C6,C7,FE,2B
```

```
1710 DATA C2,9F,C7,E5,23,CD,C8,C3,7D,32,D2,C0,C3,C6,C7,E5
1720 DATA 7E,FE,2E,20,06,23,CD,C8,C3,18,08,CD,72,C4,2A,D4
1730 DATA C0,18,03,2A,25,F9,AF,BC,28,DE,ED,5B,F3,C0,13,13
1740 DATA 7D,93,6F,C3,98,C7,E1,CD,4A,C4,AF,C9,D3,D4,AF,32
1750 DATA D3,C0,3A,D6,C0,47,21,DA,C0,7E,FE,2E,20,14,E5,23
1760 DATA CD,C8,C3,22,D4,C0,E1,CD,4A,C4,3E,FF,32,D3,C0,C3
1770 DATA 0F,C8,FE,41,38,14,FE,5B,30,10,E5,CD,72,C4,3E,FF
1780 DATA 32,D3,C0,E1,CD,4A,C4,C3,0F,C8,23,05,C2,D9,C7,21
1790 DATA DA,C0,AF,32,CE,C0,23,7E,FE,00,C8,FE,69,20,F7,23
1800 DATA 7E,FE,78,20,04,3E,DD,18,07,FE,79,C2,16,C8,3E,FD
1810 DATA E5,2A,F3,C0,77,23,22,F3,C0,3E,FF,32,CE,C0,AF,32
1820 DATA CF,C0,E1,E5,23,7E,FE,2B,C2,58,C8,23,CD,C8,C3,7D
1830 DATA 32,D0,C0,3E,FF,32,CF,C0,E1,2B,36,68,23,36,6C,23
1840 DATA 7E,FE,2B,C0,E5,23,7E,FE,29,20,FA,D1,7E,12,23,13
1850 DATA FE,00,20,F8,C9,7D,7E,7F,80,81,82,83,84,3A,DA,C0
1860 DATA 32,5A,C9,3A,DB,C0,32,5B,C9,3A,DC,C0,32,5C,C9,21
1870 DATA 38,C9,CD,CA,14,3A,25,F9,FE,00,CA,D8,C8,3D,CB,27
1880 DATA CB,27,CB,27,47,3A,DD,C0,FE,20,20,03,3A,DE,C0,21
1890 DATA 66,C9,0E,00,BE,28,04,0C,23,18,F9,78,81,47,2A,F3
1900 DATA C0,3E,CB,77,23,3A,CE,C0,FE,00,28,05,3A,D4,C0,77
1910 DATA 23,70,23,22,F3,C0,AF,C9,3A,DB,C0,FE,65,28,05,FE
1920 DATA 69,C2,6E,C9,3A,DA,C0,0E,00,FE,62,20,02,0E,40,FE
1930 DATA 72,20,02,0E,80,FE,73,20,02,0E,C0,79,FE,00,CA,6E
1940 DATA C9,2A,F3,C0,36,CB,23,3A,CE,C0,FE,00,28,05,3A,D0
1950 DATA C0,77,23,3A,D4,C0,CB,27,CB,27,CB,27,81,4F,06,00
1960 DATA 3A,E0,C0,E5,21,66,C9,BE,28,04,04,23,18,F9,E1,78
1970 DATA 81,77,23,22,F3,C0,AF,C9,FF,9E,28,28,E5,28,22,72
1980 DATA 6C,63,72,72,63,72,6C,20,72,72,20,73,6C,61,73,72
1990 DATA 61,20,20,20,73,72,6C,22,2C,22,63,70,20,22,29,F3
2000 DATA 32,29,F6,33,29,00,62,63,64,65,68,6C,28,61,3E,FF
2010 DATA C9,7A,7B,7C,7D,7E,7F,80,81,82,83,84,3A,DA,C0,FE
2020 DATA 6C,C2,E4,C9,3A,DB,C0,FE,64,C2,D4,C9,3A,E2,C0,FE
2030 DATA 2E,CA,92,CC,3A,DD,C0,21,DC,C9,CD,CC,C9,78,CB,27
2040 DATA CB,27,CB,27,C6,40,4F,21,DD,C0,7E,23,FE,2C,20,FA
2050 DATA 7E,21,DC,C9,CD,CC,C9,79,80,2A,F3,C0,77,23,3A,CE
2060 DATA C0,FE,00,28,12,3A,D4,C0,77,23,18,0B,06,00,BE,C8
2070 DATA 23,04,18,FA,3E,FF,C9,22,F3,C0,AF,C9,62,63,64,65
2080 DATA 68,6C,28,61,3A,DA,C0,32,64,CA,3A,DB,C0,32,65,CA
2090 DATA 3A,DC,C0,32,66,CA,21,42,CA,CD,CA,14,3A,25,F9,FE
2100 DATA 00,CA,D4,C9,3D,CB,27,CB,27,CB,27,C6,80,4F,3A,D3
2110 DATA C0,3C,3D,28,19,3A,CE,C0,3C,3D,20,12,3E,46,81,2A
2120 DATA F3,C0,77,23,3A,D4,C0,77,23,22,F3,C0,AF,C9,21,DC
2130 DATA C0,23,7E,FE,00,20,FA,2B,7E,FE,29,C2,B1,C9,3D,C3
2140 DATA B1,C9,FF,9E,28,28,E5,28,22,61,64,64,61,64,63,73
```

```
2150 DATA 75,62,73,62,63,61,6E,64,78,6F,72,6F,72,20,63,70
2160 DATA 20,22,2C,22,63,70,20,22,29,F3,32,29,F6,33,29,00
2170 DATA 32,D1,C0,32,D3,C0,CD,50,C3,AF,32,CF,C0,32,D3,C0
2180 DATA CD,FE,C4,CD,B2,C5,3C,3D,28,EF,CD,70,C6,3C,3D,28
2190 DATA E8,2A,F3,C0,CD,18,CC,3E,0D,DF,06,0F,C5,3E,1C,DF
2200 DATA C1,10,F9,21,DA,C0,CD,7D,69,3E,0D,DF,3E,0A,DF,3A
2210 DATA DA,C0,FE,72,20,1B,3A,DB,C0,FE,73,20,14,21,DE,C0
2220 DATA CD,C8,C3,7D,C6,C7,2A,F3,C0,77,23,22,F3,C0,C3,79
2230 DATA CA,3A,DA,C0,FE,68,20,13,3A,DB,C0,FE,61,20,0C,2A
2240 DATA F3,C0,36,76,23,22,F3,C0,C3,79,CA,3A,DA,C0,FE,69
2250 DATA 20,2C,3A,DB,C0,FE,6D,20,25,2A,F3,C0,36,ED,23,E5
2260 DATA 21,DD,C0,CD,C8,C3,7D,E1,0E,46,FE,01,20,02,0E,56
2270 DATA FE,02,20,02,0E,5E,71,23,22,F3,C0,C3,79,CA,CD,38
2280 DATA C7,CD,CE,C7,21,82,BD,0E,00,CD,EE,CB,04,05,28,0D
2290 DATA 21,4D,BF,0E,C0,CD,EE,CB,04,05,C2,93,CB,2A,F3,C0
2300 DATA 71,23,3A,D1,C0,3C,3D,28,0B,3A,D2,C0,77,23,22,F3
2310 DATA C0,C3,79,CA,3A,CF,C0,3C,3D,28,05,3A,D4,C0,77,23
2320 DATA 22,F3,C0,3A,D3,C0,3C,3D,CA,79,CA,3A,D4,C0,77,23
2330 DATA C3,77,CC,C3,5E,CC,FE,2C,CA,79,CA,FE,68,CA,79,CA
2340 DATA 3A,DC,C0,FE,74,CA,79,CA,3A,D5,C0,77,23,22,F3,C0
2350 DATA C3,79,CA,21,12,BD,0E,A0,CD,EE,CB,04,05,20,0E,2A
2360 DATA F3,C0,36,ED,23,71,23,22,F3,C0,C3,79,CA,21,80,BB
2370 DATA 0E,40,CD,EE,CB,04,05,C2,D9,CB,2A,F3,C0,36,ED,23
2380 DATA 71,23,3A,D3,C0,3C,3D,28,0A,3A,D4,C0,77,23,3A,D5
2390 DATA C0,77,23,22,F3,C0,C3,79,CA,CD,7D,C8,3C,3D,CA,79
2400 DATA CA,CD,7C,C9,3C,3D,CA,79,CA,1E,FF,C3,07,09,E5,11
2410 DATA DA,C0,06,FF,CD,0A,CC,E1,04,05,C8,0C,7E,23,FE,00
2420 DATA 20,FA,7E,06,02,FE,00,C8,18,E4,1A,BE,C0,FE,00,28
2430 DATA 04,23,13,18,F5,06,00,C9,7C,CB,3F,CB,3F,CB,3F,CB
2440 DATA 3F,CD,40,CC,7C,E6,0F,CD,40,CC,7D,CB,3F,CB,3F,CB
2450 DATA 3F,CB,3F,CD,40,CC,7D,E6,0F,CD,40,CC,C9,C3,18,CC
2460 DATA 11,4E,CC,1B,3C,13,3D,20,FC,1A,E5,DF,E1,C9,30,31
2470 DATA 32,33,34,35,36,37,38,39,41,42,43,44,45,46,3A,E0
2480 DATA C0,FE,23,C2,71,CC,3A,DA,C0,FE,69,CA,71,CC,C3,88
2490 DATA CB,3A,DE,C0,C3,76,CB,22,F3,C0,3A,DB,C0,FE,70,C2
2500 DATA 73,CB,C3,88,CB,C0,AF,C9,DD,95,96,97,98,99,9A,9B
2510 DATA 9C,9D,2A,F3,C0,36,36,23,3A,D4,C0,77,23,E5,21,E2
2520 DATA C0,CD,C8,C3,7D,E1,77,23,22,F3,C0,AF,C9,B9,BA,BB
2530 DATA BC
```

****

In the next chapter you will find the operating instructions for the SUPER ASSEMBLER.

****

CHAPTER 17

SUPER ASSEMBLER OPERATING INSTRUCTIONS

LOADING THE ASSEMBLER

SVI 328 VERSION

Type CLEAR 200,48000 followed by ENTER and then BLOAD"SA328" again followed by ENTER — press play on the tape and the assembler will load into the memory.

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 62720 (&HF500) if using a tape or above 54712 (&HD5B8) if using a disc. You may assemble programs in the memory area below the assembler but remember the following:

1) Change the original CLEAR command so that the assembled program will still be above the top of memory. For example use CLEAR 200,42000.

2) Leave sufficient space below the assembled program for the source file which starts at address 32768 (&H8000).

NOTE that if you are using the pre recorded tape version of the assembler you simply type CLOAD "SA328" 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 also that all the mini programs and source files in this book will be found on your SUPER ASSEMBLER tape.

SVI 318 VERSION

This version of the SUPER ASSEMBLER is not given in this book but is provided on the accompanying tape.
The assembler is located from address 58000 to 62480 or in HEX from &HE290 to &HF410. 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 62720 (&HF500) — You may assemble programs in the memory area below the assembler but remember the following:

1) Change the original CLEAR command (in the loader program) so that the assembled program will still be above the top of memory. For example use CLEAR 200,57000.

2) Leave sufficient space below the assembled program for the source file which starts at address 49152 (&HC000) on the unexpanded SVI 318.

****

SOURCE FILES

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 — SA328
or DEFUSR 0 = 61830 followed by ENTER — SA318

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 instrucion.

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 the assembler or your own 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	=	394d 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 ENTER — (61830 FOR SVI 318)
type DEF USR 1 = &HD000

Now type in the following source file:

SOURCE FILE 18.1

```
10 REM ! Chput demo
20 REM !
30 REM !
40 REM /.394d'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
```

Save the source file to disc or tape and then 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 AOK on the screen. This routine is used to print a character on the screen. The routine does not print a carriage return or a line feed and so the basic OK appears immediately after the character 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	=	394d 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 /.394d'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 !
 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 [.394d'Chput'!          print character rom routine
 50 REM [.3512'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	=	3512 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 [.394d'Chput'!          print character rom routine
 50 REM [.3512'Break'!          check CTRL/STOP rom routine
```

SOURCE FILE 18.4 CONTINUED

```
60 REM /.403d'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	=	403d 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 ! Chans demo
20 REM !
30 REM !
40 REM /.394d'Chput'!        print character rom routine
```

SOURCE FILE 18.5 CONTINUED

```
 50 REM [.3dca'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	=	3dca 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 [.1265'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	=	1265 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 [.1265'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 SVI 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 /.393e'Posit'!          cursor position routine
 50 REM /.1265'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 Mesg'!                  message address label
180 REM $"test message." '!     message string
190 REM db .0'!                 message end marker
195 REM ret'!                   return to basic
200 REM /'!                     end of source
```

Posit		
USE	=	locate cursor on the screen
ADDRESS	=	393e hex
ENTRY	=	X position in H Y position in L
EXIT	=	none

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 [.2c24'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	=	2c24hex
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 20.1 shows how to intitialise a repeating sound from machine code.

SOURCE FILE 20.1 20.2

```
10 REM ! sound demo          (steam train)
20 REM !
30 REM [.d000'!              assembly start address
40 REM ld a,.m7'!            psg register 7 into A
50 REM out (.88),a'!         open psg register 7
60 REM in a,(.90)'!          read current value (reg 7)
70 REM and .n11000000'!      extract bits 6 and 7
80 REM add .n00110111'!      add "on switch" noise channel A
90 REM out (.8c),a'!         initialise psg register 7
100 REM ld a,.m8'!           psg register 8 into A
110 REM out (.88),a'!        open psg register 8
120 REM ld a,.n00011111'!    volume noise channel A
130 REM out (.8c),a'!        initialise psg register 8
140 REM ld a,.m12'!          psg register 12 into A
150 REM out (.88),a'!        open psg register 12
160 REM ld a,.n00000011'!    envelope period (coarse)
170 REM out (.8c),a'!        initialise psg register 12
180 REM ld a,.m13'!          psg register 13 into A
190 REM out (.88),a'!        open psg register 13
200 REM ld a,.n00001110'!    envelope shape
210 REM out (.8c),a'!        initialise psg register 13
220 REM ret'!                return to basic
230 REM ]'!                  end of source
```

NOTE that rom routines are not used — the sound chip is accessed directly through the input/output ports. The ports used are as follows:

OUTPUT PORT	&H88 —	address latch to indicate which register is to be used.
INPUT PORT	&H90 —	used to read the value of the current register.
OUTPUT PORT	&H8C —	used to write values to current register.

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 for joystick control and

bank switching. It is therefore desirable to preserve these two bits when sound channels are enabled. Lines 60 — 80 in source file 20.2 preserve the two upper bits and enable the noise channel A of the sound chip.

****

PROGRAM FILE 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 ROM ROUTINES are used:

Eval		
USE	=	evaluate a basic expression
ADDRESS	=	14ca 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 = f793 hex
Vtyp contents = 2 for integer result.
= 3 for string result.
= 4 for single precision result.
= 8 for double precision result.

Dac is the decimal accumulator which contains the result returned by the expression evaluator:

Dac address = f923 hex
Dac contents — integer result contained in Dac + 2 and Dac + 3

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

****

Vget		
USE	=	get address of variable
ADDRESS	=	6066 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 [.14ca'Eval'!          expression evaluator
 50 REM [.6066'Vget'!          get address of variable
 60 REM [.f793'Vtyp'!          system variable value type
 70 REM [.f923'Dac'!           decimal accumulator
 80 REM [.f7f6'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 val type
```

SOURCE FILE 21.1 CONTINUED

```
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,Dac'!               point HL to Dac
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 Dac 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 file 22.1. Notice that the file has a machine code source section and a pure basic section. Assemble the file 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.

Source file 22.1 uses direct screen access through the input/output ports as follows:

OUTPUT PORT &H81 — used to set the VRAM address pointer and for writing data to the VDP registers.

OUTPUT PORT &H80 — used to send data to video ram byte located at the VRAM address pointer.

INPUT PORT &H84 — used to read data from video ram byte located at the VRAM address pointer.

NOTE that the VRAM address pointer auto increments after each read or write operation.

SOURCE FILE 22.1

```
 10 REM !                    screen 1 scroll demo
 20 REM !
 30 REM !
 40 REM [.d000'!             assembly start address
 50 REM ld a,.0'!            name table address lo byte
 60 REM out (.81),a'!        address to VDP
 70 REM ld a,.18'!           name table address hi byte
 80 REM out (.81),a'!        address to VDP
 90 REM ld hl,.d100'!        point HL to picture buffer
100 REM !
110 REM lni'!                subroutine to fetch name table
120 REM !
130 REM in a,(.84)'!         byte in
140 REM ld (hl),a'!          load into buffer
150 REM inc hl'!             increment buffer counter
160 REM ld a,.d4'!           buffer end + 1
170 REM cp h'!               is the end reached?
180 REM jr nz,lni'!          no so goto lni for next byte
190 REM !
200 REM Scrol'!              subroutine to Scroll the name table
210 REM !
220 REM ld de,.d100'!        buffer start
230 REM ld hl,.d101'!        buffer start + 1
240 REM ld bc,.001f'!        line length - 1
250 REM Loop'!               Loop label
260 REM push bc'!            save BC on stack
270 REM ld a,(de)'!          first byte in line into A register
280 REM ldir'!               move whole line one to the left
290 REM ld (de),a'!          first byte into last position
300 REM inc de'!             start of next line
310 REM inc hl'!             line start + 1
320 REM pop bc'!             recover line count
330 REM ld a,.d4'!           end check
340 REM cp h'!               is it the end?
350 REM jr nz, Loop'!        no so do it again
360 REM !
370 REM Send'!               subroutine to send to the VDP
380 REM !
390 REM di'!                 yes so prepare to send table to VDP
400 REM ld a,.0'!            name table address lo byte
410 REM out (.81),a'!        write it to the VDP
420 REM ld a,.18'!           name table address hi byte
```

SOURCE FILE 22.1 CONTINUED

```
430 REM or .40'!          set bit 6 to indicate VDP write
440 REM out (.81),a'!     write it to the VDP
450 REM ld hl,.d100'!     buffer start address into HL
460 REM Outi'!            Outi label
470 REM ld a,(hl)'!       byte into A register
480 REM out (.80),a'!     send it to VDP name table
490 REM inc hl'!          increment buffer pointer
500 REM ld a,.d3'!        end check — bottom third not sent
510 REM cp h'!            is it the end?
520 REM jr nz,Outi'!      no so go send next byte
530 REM ei'!              yes so enable interrupts
540 REM ret'!             return to basic
550 REM ]'!               end of source file
560 REM !
570 REM !
580 REM !
590 REM         basic support routine
600 REM !
610 REM !
620 REM !
630 COLOR15,8,8
640 SCREEN1
650 PSET(0,80),1
660 DRAW"e90f45e29f80e30"
670 PAINT(4,80),1
680 LINE(10,140)-(150,190),11,BF
690 LOCATE25,160
700 COLOR4
710 PRINT"SCREEN SCROLL DEMO"
720 DEFUSR2 = &HD000
730 DEFUSR3 = &HD014
740 Y = USR2(0)
750 Y = USR3(0)
760 GOTO750
```

****

The final source file in this book is a machine code version of the sprite detection routine which was presented in basic earlier.

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 22.2 is executed in the same way as file 22.1.

SOURCE FILE 22.2

```
 10 REM !                    sprite collision routine
 20 REM !
 30 REM !
 40 REM !
 50 REM [.d000'!             assembly start address
 60 REM ld b,.m33'!          sprite count into register B
 70 REM Next'!               next routine label
 80 REM dec b'!              count = count - 1
 90 REM ld a,.0'!            zero into register A
100 REM cp b'!               is count zero?
110 REM ret z'!              yes so return to basic
120 REM ld c,b'!             no so copy count into register C
130 REM sla c'!              multiply register C by 2
140 REM sla c'!              multiply by 2 again
150 REM ld a,c'!             C into A (sprite y position lo byte)
160 REM out (.81),a'!        lo byte to video chip
170 REM ld a,.1b'!           sprite y position hi byte
180 REM out (.81),a'!        hi byte to video chip
190 REM in a,(.84)'!         read in sprite y position
200 REM cp .m209'!           is it on screen?
210 REM jr nz,Test'!         yes so goto Test
220 REM jr Next'!            no so check Next sprite
230 REM !
240 REM Test'!               Test routine address label
250 REM !
260 REM push af'!            save sprite y position
270 REM ld a,c!              y position address lo byte
280 REM out (.81),a'!        lo byte to video chip
290 REM ld a,.1b'!           hi byte into A
300 REM or .40'!             set bit 6 to indicate write
310 REM out (.81),a'!        hi byte to video chip
320 REM ld a,.m209'!         "sprite off screen" into A
330 REM out (.80),a'!        write it to video chip
340 REM halt'!               force update of Vstat
350 REM ld a,(.fe3d)'!       Vstat into A
360 REM and .n00100000'!     mask unwanted bits
```

SOURCE FILE 22.2 CONTINUED

```
370 REM jr z,Found'!        goto Found if sprite not active
380 REM ld a,c'!            still active so restore
390 REM out (.81),a'!       sprite y position into
400 REM ld a,.1b'!          sprite attribute table
410 REM or .40
420 REM out (.81),a
430 REM pop af
440 REM out (.80),a
450 REM jr Next'!           go check next sprite
460 REM !
470 REM Found'!             Found routine address label
480 REM !
490 REM pop af'!            clear AF off the stack
500 REM ld hl,Varn'!        point HL to variable name
510 REM push bc'!           save sprite number (B)
520 REM call .6066'!        get location of variable
530 REM pop bc'!            recover sprite number
540 REM ex de,hl'!          point HL at variable position
550 REM ld (hl),b'!         variable lo byte
560 REM inc hl'!            increment pointer
570 REM ld (hl),.0'!        variable hi byte
580 REM ret'!               return to basic
590 REMVarn'$A'db.0']       Variable name
600 REM
610 REM
620 REM basic support program
630 REM
640 REM
650 DEFINTA-Z
660 SCREEN1
670 DEFUSR2=&HD000
680 ONSPRITEGOSUB820
690 FORX=0TO7
700 A$=A$+CHR$(255)
710 NEXT
720 SPRITE$(0)=A$
730 FORP=1TO15
740 PUTSPRITEP,(50+P*10,P*10),P,0
750 NEXT
760 PD=(INT((RND(-TIME)*180)/10))*10
770 SPRITEON
780 FORZ=-20TO255
```

SOURCE FILE 22.2 CONTINUED

```
790 PUTSPRITE0,(Z,PD),3,0
800 NEXT
810 GOTO760
820 SPRITEOFF
830 Z = USR2(0)
840 PRINTA
850 RETURN760
```

****

This book was designed to provide the reader with an introduction to machine code on the SPECTRAVIDEO — 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 65536 — 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 even (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 labels 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 SPECTRAVIDEO 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	— checks sign of a result
rst .30	F7	— gets type of variable
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

****

APPENDIX 2

SPECTRAVIDEO ROM ROUTINES

The addresses of most ROM routines can be calculated in the following manner:

1) Take the basic token (from the token table to be found earlier in the book).

2) If it is a single token calculate X = TOKEN — 129.

3) If it is a double token take the second half of the token (not the 255) and calculate X = TOKEN — 41.

4) Now calculate A = 389 + (X * 2).

5) The address of the ROM routine for the command described by the token is given by:

   ADDRESS = PEEK(A) + 256 * PEEK(A+1)

6) This procedure works for all routines with double tokens and for routines with single tokens which are less than 217.

Have fun finding the routines and figuring out how they work.

HINT — in general you point the HL register to the start of a BASIC instruction (entered in a machine code string) and call the routine. NOTE that the basic instruction must not contain the BASIC word itself.

APPENDIX 3

INPUT/OUTPUT PORT TABLE

PORT No.	I/O	DEVICE	DESCRIPTION
&H10	O	PRINTER	DATA WRITE PORT
&H11	O	PRINTER	DATA STROBE
&H12	I	PRINTER	STATUS (bit 0 = 1 if not ready)
&H20	I	MODEM	RECEIVER BUFFER REGISTER
&H20	O	MODEM	DIVISOR LATCH (LSB)
&H20	O	MODEM	TRANSMITTER BUFFER REGISTER
&H21	O	MODEM	DIVISOR LATCH (MSB)
&H21	O	MODEM	INTERRUPT ENABLE REGISTER
&H22	I	MODEM	INTERRUPT ID. REGISTER
&H23	O	MODEM	LINE CONTROL REGISTER
&H24	O	MODEM	MODEM CONTROL REGISTER
&H25	I	MODEM	LINE STATUS REGISTER
&H26	I	MODEM	MODEM STATUS REGISTER
&H28	I	RS232	RECEIVER BUFFER REGISTER
&H28	O	RS232	DIVISOR LATCH (LSB)
&H28	O	RS232	TRANSMITTER BUFFER REGISTER
&H29	O	RS232	DIVISOR LATCH (MSB)
&H29	O	RS232	INTERRUPT ENABLE REGISTER
&H2A	I	RS232	INTERRUPT ID. REGISTER
&H2B	O	RS232	LINE CONTROL REGISTER
&H2C	O	RS232	MODEM CONTROL REGISTER
&H2D	I	RS232	LINE STATUS REGISTER
&H2E	I	RS232	MODEM STATUS REGISTER
&H30	I	DISC	STATUS REGISTER
&H30	O	DISC	COMMAND REGISTER
&H31	I/O	DISC	TRACK REGISTER
&H32	I/O	DISC	SECTOR REGISTER
&H33	I/O	DISC	DATA REGISTER
&H34	I	DISC	INTRQ AND DRQ PINS
&H34	O	DISC	DISC SELECT REGISTER
&H38	O	DISC	DENSITY SELECT REGISTER
&H50	O	80 COL	REGISTER SELECT LATCH
&H51	O	80 COL	WRITE TO REGISTER
&H58	O	80 COL	CRT BANK CONTROL

INPUT/OUTPUT PORT TABLE CONTINUED

PORT No.	I/O	DEVICE	DESCRIPTION
&H80	O	VDP	VDP WRITE MODE 0
&H81	O	VDP	VDP WRITE MODE 1
&H84	I	VDP	VDP READ MODE 0
&H85	I	VDP	VDP READ MODE 1
&H88	O	PSG	REGISTER SELECT LATCH
&H8C	O	PSG	WRITE TO REGISTER
&H90	I	PSG	READ FROM REGISTER
&H96	O	PPI	WRITE PORT C
&H97	O	PPI	CONTROL WORD REGISTER
&H98	I	PPI	READ PORT A
&H99	I	PPI	READ PORT B

****

APPENDIX 4

MORE ROM ROUTINES

In appendix 2 the formula for calculating the position of the BASIC WORD MACRO ROUTINES was given — in this appendix some of the more useful PRIMITIVE ROUTINES are given.

ERAFNK

ADDRESS	&H3B86
ENTRY	NONE
EFFECT	ERASES THE FUNCTION KEY DISPLAY

DSPFNK

ADDRESS	&H3B9F
ENTRY	NONE
EFFECT	DISPLAYS THE FUNCTION KEY DEFINITIONS

RSTFNK

ADDRESS	&H3498
ENTRY	NONE
EFFECT	RESTORES THE FUNCTION KEYS TO DEFAULT STRINGS

MAPXYC

ADDRESS	&H48E9
ENTRY	X CO-ORDINATE IN BC REGISTER Y CO—ORDINATE IN DE REGISTER
EFFECT	POSITIONS THE GRAPHICS POINTER TO (X,Y)

SETC

ADDRESS	&H4988
ENTRY	GRAPHICS POINTER LOCATED AT (X,Y) REQUIRED COLOR IN ATRBYT (&HFA13)
EFFECT	SETS THE PIXEL (X,Y) TO COLOR IN ATRBYT

SETATR

ADDRESS	&H4980
ENTRY	COLOR NUMBER IN THE A REGISTER
EFFECT	SET ATRBYT TO THE COLOR IN 'A'

MORE ROM ROUTINES CONTINUED

READC

ADDRESS	&H4951
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	READ COLOR OF PIXEL (X,Y) INTO 'A'

RIGHTC

ADDRESS	&H49CF
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X+1,Y)

LEFTC

ADDRESS	&H49F8
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X−1,Y)

UPC

ADDRESS	&H4A59
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X,Y−1)

TUPC

ADDRESS	&H4A3F
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	SETS CARRY FLAG AND RETURNS IF TOP OF SCREEN IS REACHED ELSE SAME AS UPC

DOWNC

ADDRESS	&H4A2D
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	GRAPHICS POINTER TO (X,Y+1)

TDOWNC

ADDRESS	&H4A14
ENTRY	GRAPHICS POINTER AT (X,Y)
EFFECT	SETS CARRY FLAG AND RETURNS IF BOTTOM OF SCREEN IS REACHED ELSE SAME AS DOWNC

CHGCLR

ADDRESS	&H3750
ENTRY	REQUIRED FOREGROUND COLOR IN &HFA0A REQUIRED BACKGROUND COLOR IN &HFA0B REQUIRED BORDER COLOR IN &HFA0C
EFFECT	CHANGES THE SCREEN COLORS

APPENDIX 5

THE MAGIC OF SPECTRAVIDEO TAPE DIRECTORY

SIDE ONE

PROGRAM LIST 3.1	CLOAD"PL31"
PROGRAM LIST 4.1	CLOAD"PL41"
PROGRAM LIST 4.2	CLOAD"PL42"
PROGRAM LIST 4.3	CLOAD"PL43"
PROGRAM LIST 7.1	CLOAD"PL71"
PROGRAM LIST 7.2	CLOAD"PL72"
PROGRAM LIST 8.1	CLOAD"PL81"
PROGRAM LIST 10.1	CLOAD"PL101"
PROGRAM LIST 10.2	CLOAD"PL102"
PROGRAM LIST 11.1	CLOAD"PL111"
PROGRAM LIST 11.2	CLOAD"PL112"
PROGRAM LIST 11.3	CLOAD"PL113"
PROGRAM LIST 11.4	CLOAD"PL114"
PROGRAM LIST 12.1	CLOAD"PL121"
PROGRAM LIST 13.1	CLOAD"PL131"
PROGRAM LIST 14.1	CLOAD"PL141"
PROGRAM LIST 14.2	CLOAD"PL142"
SOURCE FILE 18.1	CLOAD"SF181"
SOURCE FILE 18.2	CLOAD"SF182"
SOURCE FILE 18.3	CLOAD"SF183"
SOURCE FILE 18.4	CLOAD"SF184"
SOURCE FILE 18.5	CLOAD"SF185"
SOURCE FILE 19.1	CLOAD"SF191"
SOURCE FILE 19.2	CLOAD"SF192"
SOURCE FILE 19.3	CLOAD"SF193"
SOURCE FILE 20.1	CLOAD"SF201"
SOURCE FILE 20.2	CLOAD"SF202"
PROGRAM LIST 20.3	CLOAD"PL203"
SOURCE FILE 21.1	CLOAD"SF211"
SOURCE FILE 22.1	CLOAD"SF221"
SOURCE FILE 22.2	CLOAD"SF222"
SOURCE FILE APP 6	CLOAD "SFAP6"
SOURCE FILE APP 7	CLOAD "SFAP7"

SIDE TWO

SUPER ASSEMBLER 328	CLOAD"SA328"
SUPER ASSEMBLER 328	CLOAD"SA328"
SUPER ASSEMBLER 318	CLOAD"SA318"
SUPER ASSEMBLER 318	CLOAD"SA318"

APPENDIX 6

THE BASIC STATEMENT HANDLER

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 &H0E88.

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(&HF7DD) + &H100*PEEK(&HF7DE))
```

This routine in machine code uses the following source:

```
 10 REM [.d000'!                 assembly start
 20 REM ld hl,Stap'!             Stap address into hl
 30 REM ld a,(hl)'!              first character into a
 40 REM call .0e88'!             statement handler
 50 REM ret'!                    return
 60 REM Stap
 70 REM db .m145'!               PRINT
 80 REM db .m255'db .m155'!      HEX$
 90 REM $('!                     (
100 REM db .m255'db .m151'!      PEEK
110 REM $(&HF7DD)'!              (&HF7DD)
120 REM db .m243'!               +
130 REM $&H100'!                 &H100
140 REM db .m245'!               *
150 REM db .m255'db .m151'!      PEEK
160 REM $(&HF7DE) )'!            (&HF7DE) )
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 (&HF7DD) 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 SPECTRAVIDEO — using the built in routines can save vast amounts of space in your machine code programs.

APPENDIX 7

HOOK JUMPS

In the SVI computers there are many HOOK JUMPS provided so that the programmer can "HOOK" or attach his own machine code routine to a basic ROM routine. The hook jumps are situated in the systems area of memory and each hook consists of three 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:

```
 10 REM [.d000'!          start address
 20 REM ld a,.c3'!        code for jump
 30 REM ld (.ff57),a'!    put it in the hook
 40 REM ld hl,Start'!     address for jump
 50 REM ld (.ff58),hl'!   put it in the hook
 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 .2c24'!      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 ie. "[". 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.

Hook Gone is so useful you may never need any more hooks however for completeness a full list of hooks follows:

HOOKS USED BY THE DISC SYSTEM

HDGET	&HFE7F	HINDS	&HFE82
HFPOS	&HFE8B	HLOC	&HFE9A
HBAKU	&HFE9D	HPARD	&HFEA3
HNTFL	&HFEA9	HSAVE	&HFEB5
HFILE	&HFEB8	HLOF	&HFEBB
HMERG	&HFEC7	HEOF	&HFECA
HGETP	&HFEDF	HSETF	&HFEE8
HNULO	&HFEEB	HFILO	&HFF12
HERRP	&HFF18	HBINS	&HFF2D
HWIDT	&HFF3F	HDMOT	&HFF5A
HDSKO	&HFF8A	HSETS	&HFF8D
HNAME	&HFF90	HKILL	&HFF93
HIPL	&HFF96	HCOPY	&HFF99
HDSKF	&HFF9F	HDSKI	&HFFA2
HATTR	&HFFA5		

NON DISC HOOKS

HKEYI	&HFE79	HPRTF	&HFE7C
HSCNE	&HFE85	HSNGF	&HFE88
HREAD	&HFE8E	HISRE	&HFE91
HMAIN	&HFE94	HRSLF	&HFE97
HSTKE	&HFEA0	HFRET	&HFEA6
HNTFN	&HFEAC	HCLEA	&HFEAF
HSAVD	&HFEB2	HNTPL	&HFEBE
HNODE	&HFEC1	HDOGR	&HFEC4
HPTRG	&HFECD	HNOFO	&HFED0
HPRGE	&HFED3	HBUFL	&HFED6
HCRDO	&HFED9	HOKNO	&HFEDC
HLOPD	&HFEE2	HDEVN	&HFEE5
HRETU	&HFEEE	HCLRC	&HFEF1
HLIST	&HFEF4	HRUNC	&HFEF7
HEVAL	&HFEFA	HISMI	&HFEFD
HCOMP	&HFF00	HFRQI	&HFF03
HDIRD	&HFF06	HOUTD	&HFF09
HNOTR	&HFF0C	HGEND	&HFF0F
HISFL	&HFF15	HERRF	&HFF1B
HTRMN	&HFF1E	HCRUS	&HFF21
HCRUN	&HFF24	HFINP	&HFF27
HFRME	&HFF2A	HFINI	&HFF30
HBINL	&HFF33	HFINE	&HFF36
HFING	&HFF39	HINCH	&HFF3C
HPINL	&HFF42	HQINL	&HFF45
HINLI	&HFF48	HDSKC	&HFF4B
HERAF	&HFF4E	HDSPF	&HFF51
HNEWS	&HFF54	HGONE	&HFF57
HMDMD	&HFF5D	HMDMC	&HFF60
HMDMW	&HFF63	HMDMI	&HFF66
HMDME	&HFF69	HMDMB	&HFF6C
HDIAL	&HFF6F	HRS21	&HFF72
HONGO	&HFF75	HKYCL	&HFF78
HKYEA	&HFF7B	HNMI	&HFF7E
HKEYC	&HFF81	HMON	&HFF84
HBADC	&HFF87	HCMD	&HFF9C
HMONE	&HFFA8	HINIP	&HFFAB
HCHPU	&HFFAE	HTOTE	&HFFB1

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:

1)	PRINT KEY;	Y = 8	Z = 32
2)	SELECT KEY:	Y = 8	Z = 16
3)	FUNCTION KEY 1:	Y = 7	Z = 1
4)	FUNCTION KEY 2:	Y = 7	Z = 2
5)	FUNCTION KEY 3:	Y = 7	Z = 4
6)	FUNCTION KEY 4:	Y = 7	Z = 8
7)	FUNCTION KEY 5:	Y = 7	Z = 16
8)	CTRL KEY:	Y = 6	Z = 2
9)	SHIFT KEY:	Y = 6	Z = 1
10)	LEFT GRAPH KEY:	Y = 6	Z = 4
11)	RIGHT GRAPH KEY:	Y = 6	Z = 8

Note that shifted keys produce the same values as unshifted ones — to detect between shifted and unshifted keys you should first read the shift key and then read the other key.

```
10 OUT &H96,(Y OR 16)
20 IF(INP(&H99)AND Z) <> 0 THEN 10
```

This program will loop until the key (defined by Y and Z) is pressed.

The full keyboard matrix is given below:

SPECTRAVIDEO KEYBOARD MATRIX

Y/Z	128	64	32	16	8	4	2	1
0	7	6	5	4	3	2	1	0
1	/	.	=	,	'	:	9	8
2	g	f	e	d	c	b	a	—
3	o	n	m	l	k	j	i	h
4	w	v	u	t	s	r	q	p
5	CUP	BS	]	\	[	z	y	x
6	LFT	ENT	STP	ESC	R G	L G	CTR	SFT
7	DWN	INS	CLS	F 5	F 4	F 3	F 2	F 1
8	RGT		PRT	SEL	CAP	DEL	TAB	SPC
9	7	6	5	4	3	2	1	0
10	,	.	/	*	—	+	9	8

NOTES

a) The Y value is the row number and the Z value is the column number.

b) The bottom two rows of the matrix refer to the keypad.

c) CUP, LFT, DWN and RGT refer to the cursor direction keys.

THE JOYSTICK

The joystick is another commonly used input device — the joystick direction can be read through PORT A of the PSG. Read the joystick direction as follows:

```
10 OUT&H88,&H0E
20 Z=INP (&H90)
```

Each BIT of the number Z is significant — interpret as follows:

BIT 0 — If bit 0 = 0 then joystick 1 is forward.
BIT 1 — If bit 1 = 0 then joystick 1 is backward.
BIT 2 — If bit 2 = 0 then joystick 1 is left.
BIT 3 — If bit 3 = 0 then joystick 1 is right.
BIT 4 — If bit 4 = 0 then joystick 2 is forward.
BIT 5 — If bit 5 = 0 then joystick 2 is backward.
BIT 6 — If bit 6 = 0 then joystick 2 is left.
BIT 7 — If bit 7 = 0 then joystick 2 is right.

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.

To read the joystick triggers use input port &H98.

```
10 Z=INP (&H98)
```

Bits 4 and 5 of the number Z indicate the state of the triggers:

BIT 4 — If bit 4 = 0 then the trigger on joystick 1 has been pressed.

BIT 5 — If bit 5 = 0 then the trigger on joystick 2 has been pressed.

THE MAGIC OF SPECTRAVIDEO

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 SPECTRAVIDEO" OWNERS

CODE MASTER written by B.L. BURKE

CODE MASTER creates a special environment for your SVI 328 whereby a full DISASSEMBLER and the SUPER ASSEMBLER are active simultaneously.

CODE MASTER is an extension of the SUPER ASSEMBLER and enhances its use because it includes full disassembly, a source file generator, a header reader and other advanced features which are fully explained in the accompanying manual and in this book. All this for R17.95 + G.S.T. = **R20.10.**

FREE! Postage and insurance within R.S.A.

Please send me CODE MASTER.

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.

THE MAGIC OF SPECTRAVIDEO
BY BERNARD L. BURKE

The Magic of Spectravideo 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 sections, the first of which contains useful information for the basic programmer and is essential before entering the second half of the book which deals with machine code programming.

The main features of this book include details of:

- ★ Video processor
- ★ Sound generator
- ★ Rom routines
- ★ Machine code
- ★ Super assembler

Note: All listings appearing in this book are on the accompanying tape on side one and the super assembler SV1 328/318 on side two.

Cover design: Susan Woolf

INTERSOFT