THE MSX RED BOOK

by

Avalon Software

Contents

Contents Copyright 1985 Avalon Software
Iver Lane, Cowley, Middx, UB8 2JD

MSX is a trademark of Microsoft Corp.
Z80 is a trademark of Zilog Corp.
ACADEMY is trademark of Alfred.

Introduction

Aims

This book is about MSX computers and how they work. For technical and commercial reasons MSX computer manufacturers only make a limited amount of information available to the end user about

the design of their machines. Usually this will be a fairly detailed description of Microsoft MSX BASIC together with a broad outline of the system hardware. While this level of documentation is adequate for the casual user it will inevitably prove limiting to anyone engaged in more sophisticated programming.

The aim of this book is to provide a description of the standard MSX hardware and software at a level of detail sufficient to satisfy that most demanding of users, the machine code programmer. It is not an introductory course on programming and is necessarily of a rather technical nature. It is assumed that you already possess, or intend to acquire by other means, an understanding of the Z80 Microprocessor at the machine code level. As there are so many general purpose books already in existence about the Z80 any description of its characteristics would simply duplicate widely available information.

Organization

The MSX Standard specifies the following as the major functional components in any MSX computer:

1. Zilog Z80 Microprocessor
2. Intel 8255 Programmable Peripheral Interface
3. Texas 9929 Video Display Processor
4. General Instrument 8910 Programmable Sound Generator
5. 32 KB MSX BASIC ROM
6. 8 KB RAM minimum

Although there are obviously a great many additional components involved in the design of an MSX computer they are all small-scale, non-programmable ones and therefore "invisible" to the user. Manufacturers generally have considerable freedom in the selection of these small-scale components. The programmable components cannot be varied and therefore all MSX machines are identical as far as the programmer is concerned.

Chapters 1, 2 and 3 describe the operation of the Programmable Peripheral Interface, Video Display Processor and Programmable Sound Generator respectively. These three devices provide the interface between the Z80 and the peripheral hardware on a standard MSX machine. All occupy positions on the Z80 I/O (Input/Output) Bus.

Chapter 4 covers the software contained in the first part of the MSX ROM. This section of the ROM is concerned with controlling the machine hardware at the fine detail level and is known as the ROM BIOS (Basic Input Output System). It is structured in such a way that most of the functions a machine code programmer requires, such as keyboard and video drivers, are readily available.

Chapter 5 describes the software contained in the remainder of the ROM, the Microsoft MSX BASIC Interpreter. Although this is largely a text-driven program, and consequently of less use to the programmer, a close examination reveals many points not documented by manufacturers.

Chapter 6 is concerned with the organization of system memory. Particular attention is paid to the Workspace Area, that section of RAM from F380H to FFFFH, as this is used as a scratchpad by the BIOS and the BASIC Interpreter and contains much information of use to any application program.

Chapter 7 gives some examples of machine code programs that make use of ROM features to minimize design effort.

It is believed that this book contains zero defects, if you know otherwise the author would be delighted to hear from you. This book is dedicated to the Walking Nightmare.

1. Programmable Peripheral Interface

The 8255 PPI is a general purpose parallel interface device configured as three eight bit data ports, called A, B and C, and a mode port. It appears to the Z80 as four I/O ports through which the keyboard, the memory switching hardware, the cassette motor, the cassette output, the Caps Lock LED and the Key Click audio output can be controlled. Once the PPI has been initialized access to a particular piece of hardware just involves writing to or reading the relevant I/O port.

PPI Port A (I/O Port A8H)

7 6	5 4	3 2	1 0
Page 3 PSLOT# C000 - FFFF	Page 2 PSLOT# 8000 - BFFF	Page 1 PSLOT# 4000 - 7FFF	Page 0 PSLOT# 0000 - 3FFF

Figure 1: Primary Slot Register

This output port, known as the Primary Slot Register in MSX terminology, is used to control the memory switching hardware. The Z80 Microprocessor can only access 64 KB of memory directly. This limitation is currently regarded as too restrictive and several of the newer personal computers employ methods to overcome it.

MSX machines can have multiple memory devices at the same address and the Z80 may switch in any one of them as required. The processor address space is regarded as being duplicated "sideways" into four separate 64 KB areas, called Primary Slots 0 to 3, each of which receives its own slot select signal alongside the normal Z80 bus signals. The contents of the Primary Slot Register determine which slot select signal is active and therefore which Primary Slot is selected.

To increase flexibility each 16 KB "page" of the Z80 address space may be selected from a different Primary Slot. As shown in Figure 1 two bits of the Primary Slot Register are required to define the Primary Slot number for each page.

The first operation performed by the MSX ROM at power-up is to search through each slot for RAM in pages 2 and 3 (8000H to FFFFH). The Primary Slot Register is then set so that the relevant slots are selected thus making the RAM permanently available. The memory configuration of any MSX machine can be determined by displaying the Primary Slot Register setting with the BASIC statement:

```
PRINT RIGHT$("0000000"+BIN$(INP(&HA8)),8)
```

As an example "10100000" would be produced on a Toshiba HX10 where pages 3 and 2 (the RAM) both come from Primary Slot 2 and pages 1 and 0 (the MSX ROM) from Primary Slot 0. The MSX ROM must always be placed in Primary Slot 0 by a manufacturer as this is the slot selected by the hardware at power-up. Other memory devices, RAM and any additional ROM, may be placed in any slot by a manufacturer.

A typical UK machine will have one Primary Slot containing the MSX ROM, one containing 64 KB of RAM and two slots brought out to external connectors. Most Japanese machines have a cartridge type connector on each of these external slots but UK machines usually have one cartridge connector and one IDC connector.

Expanders

System memory can be increased to a theoretical maximum of sixteen 64 KB areas by using expander interfaces. An expander plugs into any Primary Slot to provide four 64 KB Secondary Slots, numbered 0 to 3, instead of one primary one. Each expander has its own local hardware, called a Secondary Slot Register, to select which of the Secondary Slots should appear in the Primary Slot. As before pages can be selected from different Secondary Slots.

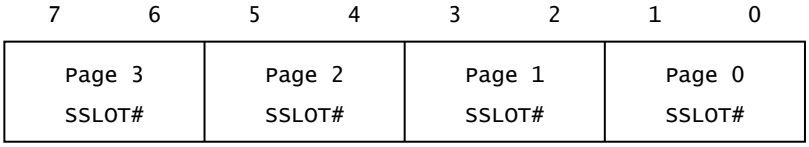


Figure 2: Secondary Slot Register

Each Secondary Slot Register, while actually being an eight bit read/write latch, is made to appear as memory location FFFFH of its Primary Slot by the expander hardware. In order to gain access to this location on a particular expander it will usually be necessary to first switch page 3 (C000H to FFFFH) of that Primary Slot into the processor address space. The Secondary Slot Register can then be modified and, if necessary, page 3 restored to its original Primary Slot setting. Accessing memory in expanders can become rather a convoluted process.

It is apparent that there must be some way of determining whether a Primary Slot contains ordinary RAM or an expander in order to access it properly. To achieve this the Secondary Slot Registers are designed to invert their contents when read back. During the power-up RAM search memory location FFFFH of each Primary Slot is examined to determine whether it behaves normally or whether the slot contains an expander. The results of these tests are stored in the Workspace Area system resource map EXPTBL for later use. This is done at power-up because of the difficulty in performing tests when the Secondary Slot Registers actually contain live settings.

Memory switching is obviously an area demanding extra caution, particularly with the hierarchical mechanisms needed to control expanders. Care must be taken to avoid switching out the page in which a program is running or, if it is being used, the page containing the stack. There are a number of standard routines available to the machine code programmer in the BIOS section of the MSX ROM to simplify the process.

The BASIC Interpreter itself has four methods of accessing extension ROMs. The first three of these are for use with machine code ROMs placed in page 1 (4000H to 7FFFH), they are:

1. Hooks (Chapter 6).
2. The "`CALL`" statement (Chapter 5).
3. Additional device names (Chapter 5).

The BASIC Interpreter can also execute a BASIC program ROM detected in page 2 (8000H to BFFFH) during the power-up ROM search. What the BASIC Interpreter cannot do is use any RAM hidden behind other memory devices. This limitation is a reflection of the difficulty in converting an established program to take advantage of newer, more complex machines. A similar situation exists with the version of Microsoft BASIC available on the IBM PC. Out of a 1 MB memory space only 64 KB can be used for program storage.

PPI Port B (I/O Port A9H)

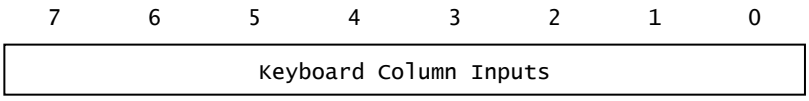


Figure 3

This input port is used to read the eight bits of column data from the currently selected row of the keyboard. The MSX keyboard is a software scanned eleven row by eight column matrix of normally open switches. Current machines usually only have keys in rows zero to eight. Conversion of key depressions into character codes is performed by the MSX ROM interrupt handler, this process is described in Chapter 4.

PPI Port C (I/O Port AAH)

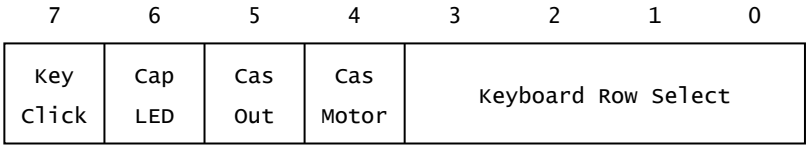


Figure 4

This output port controls a variety of functions. The four Keyboard Row Select bits select which of the eleven keyboard rows, numbered from 0 to 10, is to be read in by PPI Port B.

The Cas Motor bit determines the state of the cassette motor relay: 0=On, 1=Off.

The Cas Out bit is filtered and attenuated before being taken to the cassette DIN socket as the MIC signal. All cassette tone generation is performed by software.

The Cap LED bit determines the state of the Caps Lock LED: 0=On, 1=Off.

The Key Click output is attenuated and mixed with the audio output from the Programmable Sound Generator. To actually generate a sound this bit should be flipped on and off.

Note that there are standard routines in the ROM BIOS to access all of the functions available with this port. These should be used in preference to direct manipulation of the hardware if at all possible.

PPI Mode Port (I/O Port ABH)

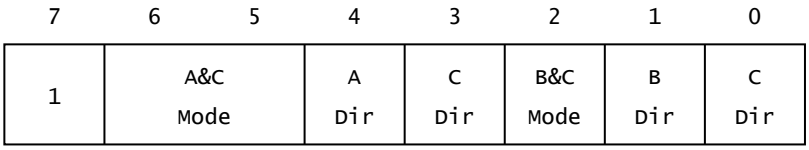


Figure 5: PPI Mode Selection

This port is used to set the operating mode of the PPI. As the MSX hardware is designed to work in one particular configuration only this port should not be modified under any circumstances. Details are given for completeness only.

Bit 7 must be 1 in order to alter the PPI mode, when it is 0 the PPI performs the single bit set/reset function shown in Figure 6.

The A&C Mode bits determine the operating mode of Port A and the upper four bits only of Port C: 00=Normal Mode (MSX), 01=Strobed Mode, 10=Bidirectional Mode

The A Dir mode determines the direction of Port A: 0=Output (MSX), 1=Input.

The C Dir bit determines the direction of the upper four bits only of Port C: 0=Output (MSX), 1=Input.

The B&C Mode bits determine the operating mode of Port B and the lower four bits only of Port C: 0=Normal Mode (MSX), 1=Strobed Mode.

The B Dir bit determines the direction of Port B: 0=Output, 1=Input (MSX).

The C Dir bit determines the direction of the lower four bits only of Port C: 0=Output (MSX), 1=Input

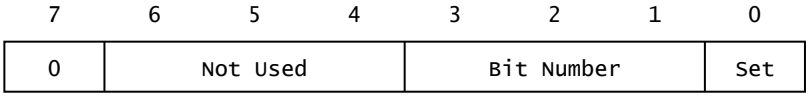


Figure 6: PPI Bit Set/Reset

The PPI Mode Port can be used to directly set or reset any bit of Port C when bit 7 is 0. The Bit Number, from 0 to 7, determines which bit is to be affected. Its new value is determined by the Set/Reset bit: 0=Reset, 1=Set. The advantage of this mode is that a single output can be easily modified. As an example the Caps Lock LED may be turned on with the BASIC statement `OUT &HAB,12` and off with the statement `OUT &HAB,13`.

2. Video Display Processor

The 9929 VDP contains all the circuitry necessary to generate the video display. It appears to the Z80 as two I/O ports called the Data Port and the Command Port. Although the VDP has its own 16 KB of VRAM (Video RAM), the contents of which define the screen image, this cannot be directly accessed by the Z80. Instead it must use the two I/O ports to modify the VRAM and to set the various VDP operating conditions.

Data Port (I/O Port 98H)

The Data Port is used to read or write single bytes to the VRAM. The VDP possesses an internal address register pointing to a location in the VRAM. Reading the Data Port will input the byte from this VRAM location while writing to the Data Port will store a byte there. After a read or write the address register is automatically incremented to point to the next VRAM location. Sequential bytes can be accessed simply by continuous reads or writes to the Data Port.

Command Port (I/O Port 99H)

The Command Port is used for three purposes:

1. To set up the Data Port address register.

2. To read the VDP Status Register.
3. To write to one of the VDP Mode Registers.

Address Register

The Data Port address register must be set up in different ways depending on whether the subsequent access is to be a read or a write. The address register can be set to any value from 0000H to 3FFFH by first writing the LSB (Least Significant Byte) and then the MSB (Most Significant Byte) to the Command Port. Bits 6 and 7 of the MSB are used by the VDP to determine whether the address register is being set up for subsequent reads or writes as follows:

Read	xxxxxxxx	00xxxxxx
Write	xxxxxxxx	01xxxxxx

Figure 7: VDP Address Setup

It is important that no other accesses are made to the VDP in between writing the LSB and the MSB as this will upset its synchronization. The MSX ROM interrupt handler is continuously reading the VDP Status Register as a background task so interrupts should be disabled as necessary.

VDP Status Register

Reading the Command Port will input the contents of the VDP Status Register. This contains various flags as below:

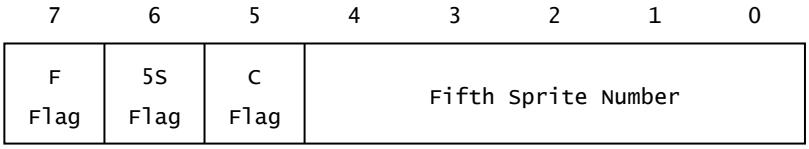


Figure 8: VDP Status Register

The Fifth Sprite Number bits contain the number (0 to 31) of the sprite triggering the Fifth Sprite Flag.

The Coincidence Flag is normally 0 but is set to 1 if any sprites have one or more overlapping pixels. Reading the Status Register will reset this flag to a 0. Note that coincidence is only checked as each pixel is generated during a video frame, on a UK machine this is every 20 ms. If fast moving sprites pass over each other between checks then no coincidence will be flagged.

The Fifth Sprite Flag is normally 0 but is set to 1 when there are more than four sprites on any pixel line. Reading the Status Register will reset this flag to a 0.

The Frame Flag is normally 0 but is set to a 1 at the end of the last active line of the video frame. For UK machines with a 50 Hz frame rate this will occur every 20 ms. Reading the Status register will reset

this flag to a 0. There is an associated output signal from the VDP which generates Z80 interrupts at the same rate, this drives the MSX ROM interrupt handler.

VDP Mode Registers

The VDP has eight write-only registers, numbered 0 to 7, which control its general operation. A particular register is set by first writing a data byte then a register selection byte to the Command Port. The register selection byte contains the register number in the lower three bits: 10000RRR. As the Mode Registers are write-only, and cannot be read, the MSX ROM maintains an exact copy of the eight registers in the Workspace Area of RAM (Chapter 6). Using the MSX ROM standard routines for VDP functions ensures that this register image is correctly updated.

Mode Register 0

7	6	5	4	3	2	1	0
0	0	0	0	0	0	M3	EV

Figure 9

The External VDP bit determines whether external VDP input is to be enabled or disabled: 0=Disabled, 1=Enabled.

The M3 bit is one of the three VDP mode selection bits, see Mode Register 1.

Mode Register 1

7	6	5	4	3	2	1	0
4/16K	Blank	IE	M1	M2	0	Size	Mag

Figure 10

The Magnification bit determines whether sprites will be normal or doubled in size: 0=Normal, 1=Doubled.

The Size bit determines whether each sprite pattern will be 8x8 bits or 16x16 bits: 0=8x8, 1=16x16.

The M1 and M2 bits determine the VDP operating mode in conjunction with the M3 bit from Mode Register 0:

```
M1 M2 M3
0  0  0  32x24 Text Mode
0  0  1  Graphics Mode
```

```
0  1  0  Multicolour Mode
1  0  0  40x24 Text Mode
```

The Interrupt Enable bit enables or disables the interrupt output signal from the VDP: 0=Disable, 1=Enable.

The Blank bit is used to enable or disable the entire video display: 0=Disable, 1=Enable. When the display is blanked it will be the same colour as the border.

The 4/16K bit alters the VDP VRAM addressing characteristics to suit either 4 KB or 16 KB chips: 0=4 KB, 1=16 KB.

Mode Register 2

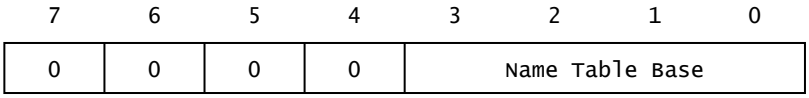


Figure 11

Mode Register 2 defines the starting address of the Name Table in the VDP VRAM. The four available bits only specify positions 00BB BB00 0000 0000 of the full address so register contents of 0FH would result in a base address of 3C00H.

Mode Register 3

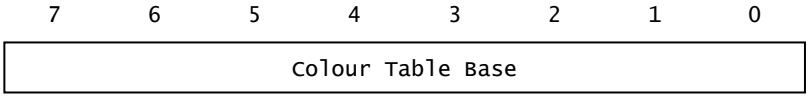


Figure 12

Mode Register 3 defines the starting address of the Colour Table in the VDP VRAM. The eight available bits only specify positions 00BB BBBB BB00 0000 of the full address so register contents of FFH would result in a base address of 3FC0H. In Graphics Mode only bit 7 is effective thus offering a base of 0000H or 2000H. Bits 0 to 6 must be 1.

Mode Register 4

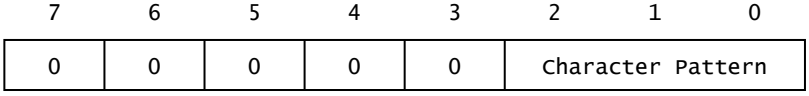


Figure 13

Mode Register 4 defines the starting address of the Character Pattern Table in the VDP VRAM. The three available bits only specify positions 00BB B000 0000 0000 of the full address so register contents of 07H would result in a base address of 3800H. In Graphics Mode only bit 2 is effective thus offering a base of 0000H or 2000H. Bits 0 and 1 must be 1.

Mode Register 5

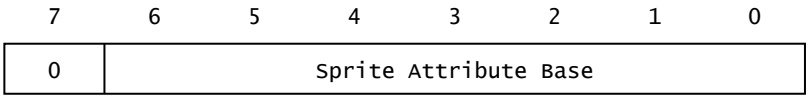


Figure 14

Mode Register 5 defines the starting address of the Sprite Attribute Table in the VDP VRAM. The seven available bits only specify positions 00BB BBBB B000 0000 of the full address so register contents of 7FH would result in a base address of 3F80H.

Mode Register 6

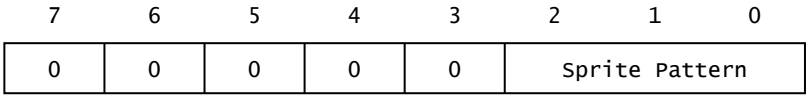


Figure 15

Mode Register 6 defines the starting address of the Sprite Pattern Table in the VDP VRAM. The three available bits only specify positions 00BB B000 0000 0000 of the full address so register contents of 07H would result in a base address of 3800H.

Mode Register 7

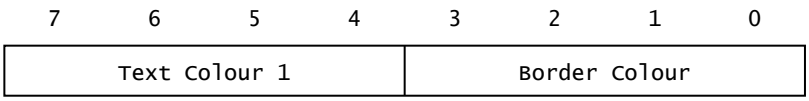


Figure 16

The Border Colour bits determine the colour of the region surrounding the active video area in all four VDP modes. They also determine the colour of all 0 pixels on the screen in 40x24 Text Mode. Note that the border region actually extends across the entire screen but will only become visible in the active area if the overlying pixel is transparent.

The Text Colour 1 bits determine the colour of all 1 pixels in 40x24 Text Mode. They have no effect in the other three modes where greater flexibility is provided through the use of the Colour Table. The VDP colour codes are:

```
 0 Transparent   4 Dark Blue      8 Red              12 Dark Green
 1 Black         5 Light Blue     9 Bright Red       13 Purple
 2 Green         6 Dark Red      10 Yellow           14 Grey
 3 Light Green   7 Sky Blue      11 Light Yellow     15 White
```

Screen Modes

The VDP has four operating modes, each one offering a slightly different set of capabilities. Generally speaking, as the resolution goes up the price to be paid in VRAM size and updating complexity also increases. In a dedicated application these associated hardware and software costs are important considerations. For an MSX machine they are irrelevant, it therefore seems a pity that a greater attempt was not made to standardize on one particular mode. The Graphics Mode is capable of adequately performing all the functions of the other modes with only minor reservations.

An added difficulty in using the VDP arises because insufficient allowance was made in its design for the overscanning used by most televisions. The resulting loss of characters at the screen edges has forced all the video-related MSX software into being based on peculiar screen sizes. UK machines normally use only the central thirty-seven characters available in 40x24 Text Mode. Japanese machines, with NTSC (National Television Standards Committee) video outputs, use the central thirty-nine characters.

The central element in the VDP, from the programmer's point of view, is the Name Table. This is a simple list of single- byte character codes held in VRAM. It is 960 bytes long in 40x24 Text Mode, 768 bytes long in 32x24 Text Mode, Graphics Mode and Multicolour Mode. Each position in the Name Table corresponds to a particular location on the screen.

During a video frame the VDP will sequentially read every character code from the Name Table, starting at the base. As each character code is read the corresponding 8x8 pattern of pixels is looked up in the Character Pattern Table and displayed on the screen. The appearance of the screen can thus be modified by either changing the character codes in the Name Table or the pixel patterns in the Character Pattern Table.

Note that the VDP has no hardware cursor facility, if one is required it must be software generated.

40x24 Text Mode

The Name Table occupies 960 bytes of VRAM from 0000H to 03BFH:

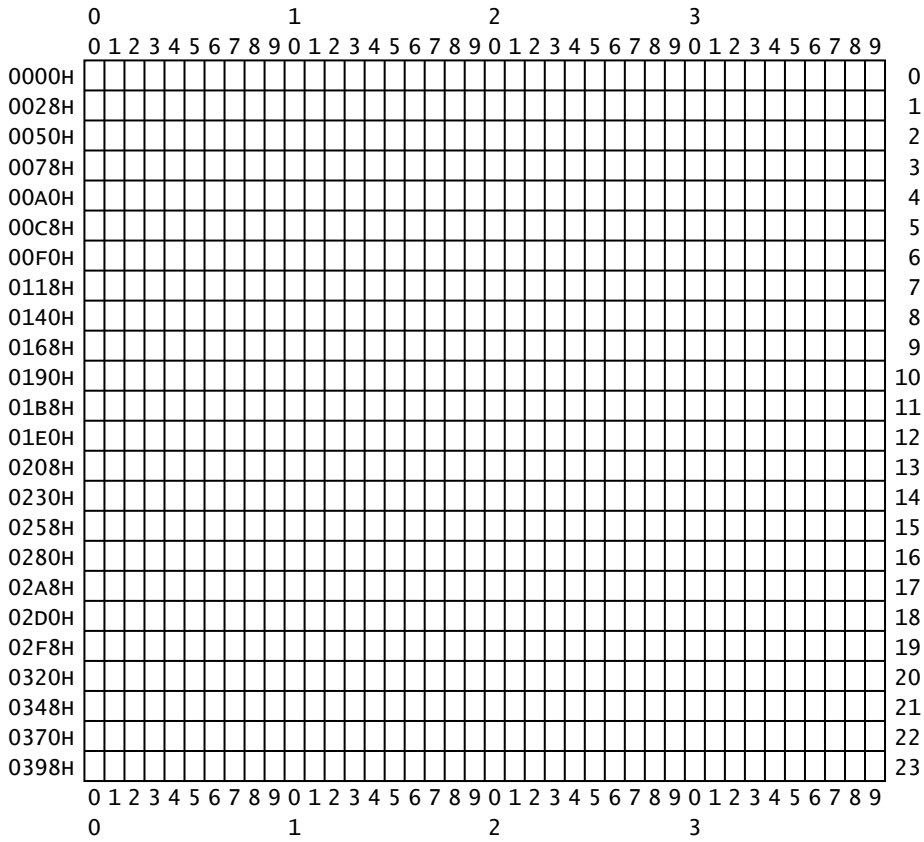


Figure 17: 40x24 Name Table

Pattern Table occupies 2 KB of VRAM from 0800H to 0FFFH. Each eight byte block contains the pixel pattern for a character code:

```
0 0 1 0 0 0 0 0  Byte 0
0 1 0 1 0 0 0 0  Byte 1
1 0 0 0 1 0 0 0  Byte 2
1 0 0 0 1 0 0 0  Byte 3
1 1 1 1 1 0 0 0  Byte 4
1 0 0 0 1 0 0 0  Byte 5
1 0 0 0 1 0 0 0  Byte 6
0 0 0 0 0 0 0 0  Byte 7
```

Figure 18: Character Pattern Block (No. 65 shown = 'A')

The first block contains the pattern for character code 0, the second the pattern for character code 1 and so on to character code 255. Note that only the leftmost six pixels are actually displayed in this mode. The colours of the 0 and 1 pixels in this mode are defined by VDP Mode Register 7, initially they are blue and white.

32x24 Text Mode

The Name Table occupies 768 bytes of VRAM from 1800H to 1AFFH. As in 40x24 Text Mode normal operation involves placing character codes in the required position in the table. The "`VPOKE`" statement may be used to attain familiarity with the screen layout:

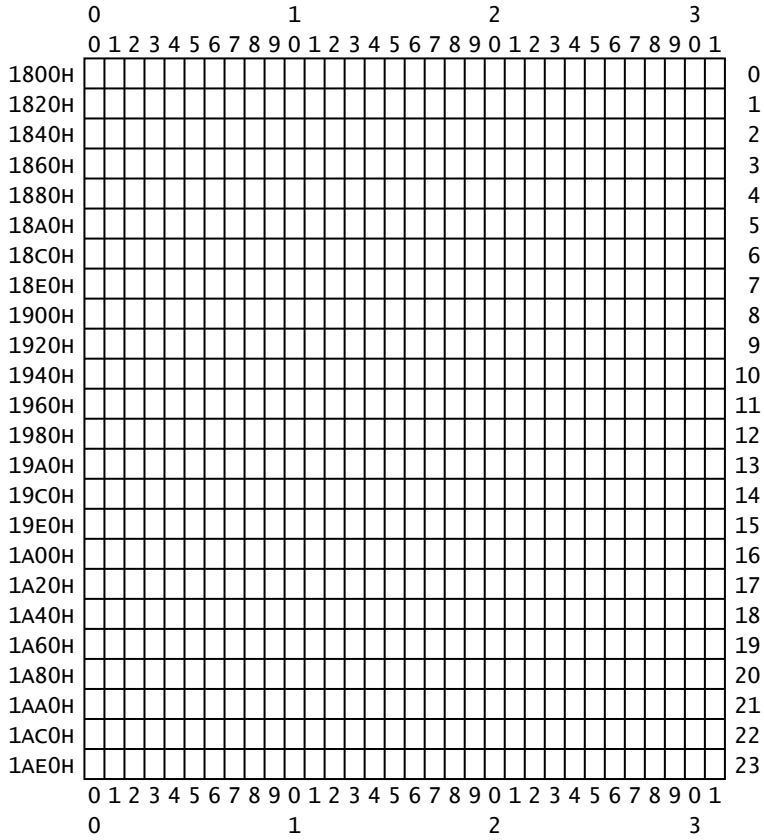


Figure 19: 32x24 Name Table

The Character Pattern Table occupies 2 KB of VRAM from 0000H to 07FFH. Its structure is the same as in 40x24 Text Mode, all eight pixels of an 8x8 pattern are now displayed.

The border colour is defined by VDP Mode Register 7 and is initially blue. An additional table, the Colour Table, determines the colour of the 0 and 1 pixels. This occupies thirty-two bytes of VRAM from 2000H to 201FH. Each entry in the Colour Table defines the 0 and 1 pixel colours for a group of eight character codes, the lower four bits defining the 0 pixel colour, the upper four bits the 1 pixel colour. The first entry in the table defines the colours for character codes 0 to 7, the second for character codes 8 to 15 and so on for thirty-two entries. The MSX ROM initializes all entries to the same value, blue and white, and provides no facilities for changing individual ones.

Graphics Mode

The Name Table occupies 768 bytes of VRAM from 1800H to 1AFFH, the same as in 32x24 Text Mode. The table is initialized with the character code sequence 0 to 255 repeated three times and is then left untouched, in this mode it is the Character Pattern Table which is modified during normal operation.

The Character Pattern Table occupies 6 KB of VRAM from 0000H to 17FFH. While its structure is the same as in the text modes it does not contain a character set but is initialized to all 0 pixels. The first 2 KB of the Character Pattern Table is addressed by the character codes from the first third of the Name Table, the second 2 KB by the central third of the Name Table and the last 2 KB by the final third of the Name Table. Because of the sequential pattern in the Name Table the entire Character Pattern Table is read out linearly during a video frame. Setting a point on the screen involves working out

where the corresponding bit is in the Character Pattern Table and turning it on. For a BASIC program to convert X,Y coordinates to an address see the MAPXYC standard routine in Chapter 4.

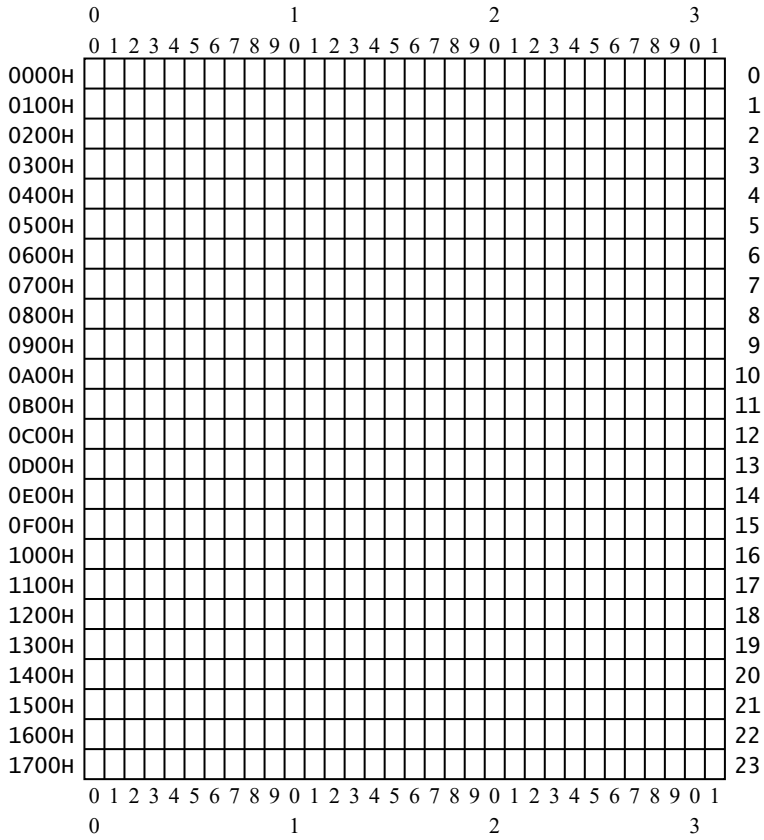


Figure 20: Graphics Character Pattern Table

The border colour is defined by VDP Mode Register 7 and is initially blue. The Colour Table occupies 6 KB of VRAM from 2000H to 37FFH. There is an exact byte-to-byte mapping from the Character Pattern Table to the Colour Table but, because it takes a whole byte to define the 0 pixel and 1 pixel colours, there is a lower resolution for colours than for pixels. The lower four bits of a Colour Table entry define the colour of all the 0 pixels on the corresponding eight pixel line. The upper four bits define the colour of the 1 pixels. The Colour Table is initialized so that the 0 pixel colour and the 1 pixel colour are blue for the entire table. Because both colours are the same it will be necessary to alter one colour when a bit is set in the Character Pattern Table.

Multicolour Mode

The Name Table occupies 768 bytes of VRAM from 0800H to 0AFFH, the screen mapping is the same as in 32x24 Text Mode. The table is initialized with the following character code pattern:

```
00H to 1FH (Repeated four times)
20H to 3FH (Repeated four times)
40H to 5FH (Repeated four times)
60H to 7FH (Repeated four times)
80H to 9FH (Repeated four times)
A0H to BFH (Repeated four times)
```

As with Graphics Mode this is just a character code "driver" pattern, it is the Character Pattern Table which is modified during normal operation.

The Character Pattern table occupies 1536 bytes of VRAM from 0000H to 05FFH. As in the other modes each character code maps onto an eight byte block in the Character Pattern Table. Because of the lower resolution in this mode only two bytes of the pattern block are actually needed to define an 8x8 pattern:

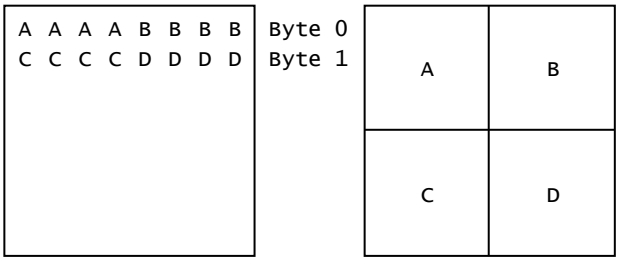


Figure 21: Multicolour Pattern Block

As can be seen from Figure 21 each four bit section of the two byte block contains a colour code and thus defines the colour of a quadrant of the 8x8 pixel pattern. So that the entire eight bytes of the pattern block can be utilized a given character code will use a different two byte section depending upon the character code's screen location (i.e. its position in the Name Table):

```
Video row 0, 4, 8, 12, 16, 20   Uses bytes 0 and 1
Video row 1, 5, 9, 13, 17, 21   Uses bytes 2 and 3
Video row 2, 6, 10, 14, 18, 22  Uses bytes 4 and 5
Video row 3, 7, 11, 15, 19, 23  Uses bytes 6 and 7
```

When the Name Table is filled with the special driver sequence of character codes shown above the Character Pattern Table will be read out linearly during a video frame:

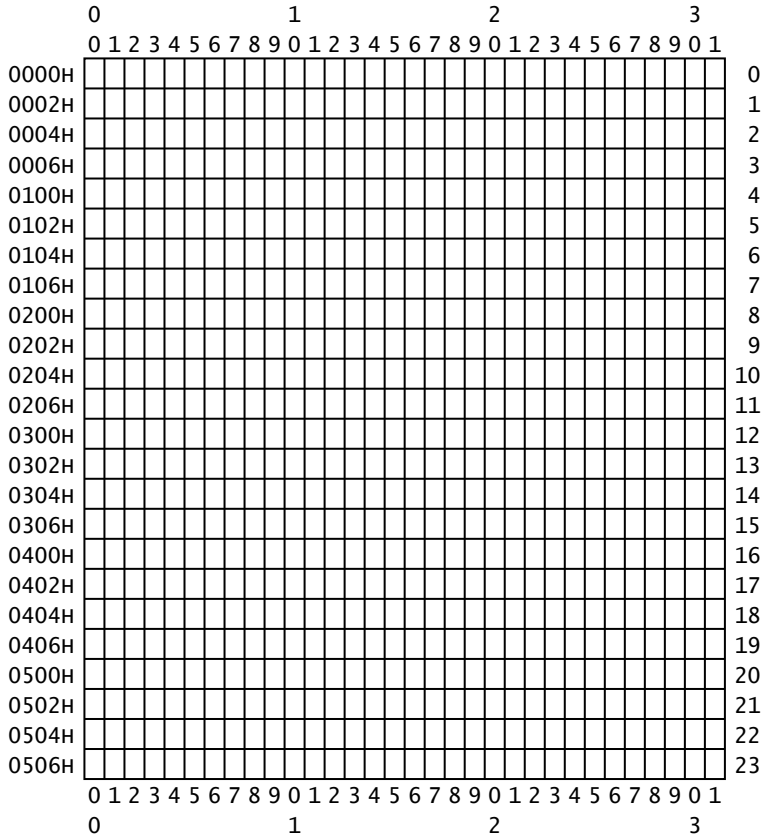


Figure 22: Multicolour Character Pattern Table

The border colour is defined by VDP Mode Register 7 and is initially blue. There is no separate Colour Table as the colours are defined directly by the contents of the Character Pattern Table, this is initially filled with blue.

Sprites

The VDP can control thirty-two sprites in all modes except 40x24 Text Mode. Their treatment is identical in all modes and independent of any character-orientated activity.

The Sprite Attribute Table occupies 128 bytes of VRAM from 1B00H to 1B7FH. The table contains thirty-two four byte blocks, one for each sprite. The first block controls sprite 0 (the "top" sprite), the second controls sprite 1 and so on to sprite 31. The format of each block is as below:

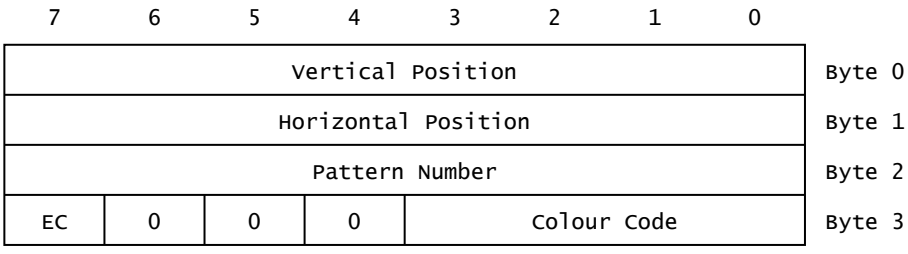


Figure 23: Sprite Attribute Block

Byte 0 specifies the vertical (Y) coordinate of the top-left pixel of the sprite. The coordinate system runs from -1 (FFH) for the top pixel line on the screen down to 190 (BEH) for the bottom line. Values

less than -1 can be used to slide the sprite in from the top of the screen. The exact values needed will obviously depend upon the size of the sprite. Curiously there has been no attempt in MSX BASIC to reconcile this coordinate system with the normal graphics range of Y=0 to 191. As a consequence a sprite will always be one pixel lower on the screen than the equivalent graphic point. Note that the special vertical coordinate value of 208 (D0H) placed in a sprite attribute block will cause the VDP to ignore all subsequent blocks in the Sprite Attribute Table. Effectively this means that any lower sprites will disappear from the screen.

Byte 1 specifies the horizontal (X) coordinate of the top- left pixel of the sprite. The coordinate system runs from 0 for the leftmost pixel to 255 (FFH) for the rightmost. As this coordinate system provides no mechanism for sliding a sprite in from the left a special bit in byte 3 is used for this purpose, see below.

Byte 2 selects one of the two hundred and fifty-six 8x8 bit patterns available in the Sprite Pattern Table. If the Size bit is set in VDP Mode Register 1, resulting in 16x16 bit patterns occupying thirty-two bytes each, the two least significant bits of the pattern number are ignored. Thus pattern numbers 0, 1, 2 and 3 would all select pattern number 0.

In Byte 3 the four Colour Code bits define the colour of the 1 pixels in the sprite patterns, 0 pixels are always transparent. The Early Clock bit is normally 0 but will shift the sprite thirty-two pixels to the left when set to 1. This is so that sprites can slide in from the left of the screen, there being no spare coordinates in the horizontal direction.

The Sprite Pattern Table occupies 2 KB of VRAM from 3800H to 3FFFH. It contains two hundred and fifty-six 8x8 pixel patterns, numbered from 0 to 255. If the Size bit in VDP Mode Register 1 is 0, resulting in 8x8 sprites, then each eight byte sprite pattern block is structured in the same way as the character pattern block shown in Figure 18. If the Size bit is 1, resulting in 16x16 sprites, then four eight byte blocks are needed to define the pattern as below:

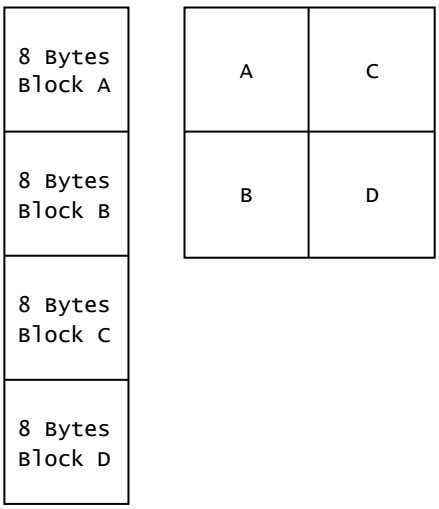


Figure 24: 16x16 Sprite Pattern Block

3. Programmable Sound Generator

As well as controlling three sound channels the 8910 PSG contains two eight bit data ports, called A and B, through which it interfaces the joysticks and the cassette input. The PSG appears to the Z80 as three I/O ports called the Address Port, the Data Write Port and the Data Read Port.

Address Port (I/O port A0H)

The PSG contains sixteen internal registers which completely define its operation. A specific register is selected by writing its number, from 0 to 15, to this port. Once selected, repeated accesses to that register may be made via the two data ports.

Data Write Port (I/O port A1H)

This port is used to write to any register once it has been selected by the Address Port.

Data Read Port (I/O port A2H)

This port is used to read any register once it has been selected by the Address Port.

Registers 0 and 1

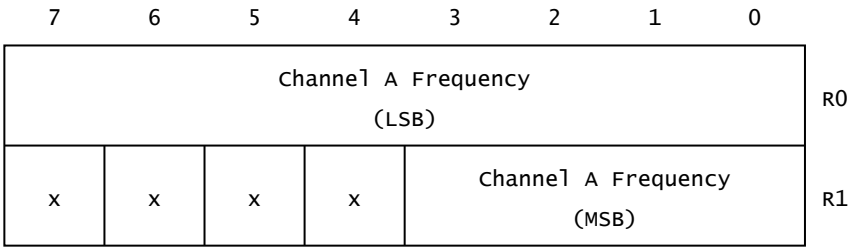


Figure 25

These two registers are used to define the frequency of the Tone Generator for Channel A. Variable frequencies are produced by dividing a fixed master frequency with the number held in Registers 0 and 1, this number can be in the range 1 to 4095. Register 0 holds the least significant eight bits and Register 1 the most significant four. The PSG divides an external 1.7897725 MHz frequency by sixteen to produce a Tone Generator master frequency of 111,861 Hz. The output of the Tone Generator can therefore range from 111,861 Hz (divide by 1) down to 27.3 Hz (divide by 4095). As an example to produce a middle "`A`" (440 Hz) the divider value in Registers 0 and 1 would be 254.

Registers 2 and 3

These two registers control the Channel B Tone Generator as for Channel A.

Registers 4 and 5

These two registers control the Channel C Tone Generator as for Channel A.

Register 6

7	6	5	4	3	2	1	0
x	x	x	Noise Frequency				

Figure 26

In addition to three square wave Tone Generators the PSG contains a single Noise Generator. The fundamental frequency of the noise source can be controlled in a similar fashion to the Tone Generators. The five least significant bits of Register 6 hold a divider value from 1 to 31. The Noise Generator master frequency is 111,861 Hz as before.

Register 7

7	6	5	4	3	2	1	0
Port B Dir	Port A Dir	C Noise	B Noise	A Noise	C Tone	B Tone	A Tone

Figure 27

This register enables or disables the Tone Generator and Noise Generator for each of the three channels: 0=Enable 1=Disable. It also controls the direction of interface ports A and B, to which the joysticks and cassette are attached: 0=Input, 1=Output. Register 7 must always contain 10xxxxxx or possible damage could result to the PSG, there are active devices connected to its I/O pins. The BASIC "`SOUND`" statement will force these bits to the correct value for Register 7 but there is no protection at the machine code level.

Register 8

7	6	5	4	3	2	1	0
x	x	x	Mode	Channel A Amplitude			

Figure 28

The four Amplitude bits determine the amplitude of Channel A from a minimum of 0 to a maximum of 15. The Mode bit selects either fixed or modulated amplitude: 0=Fixed, 1=Modulated. When modulated amplitude is selected the fixed amplitude value is ignored and the channel is modulated by the output from the Envelope Generator.

Register 9

This register controls the amplitude of Channel B as for Channel A.

Register 10

This register controls the amplitude of Channel C as for Channel A.

Registers 11 and 12

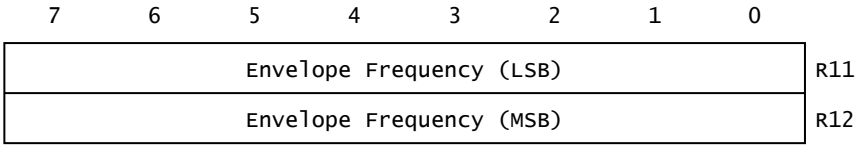


Figure 29

These two registers control the frequency of the single Envelope Generator used for amplitude modulation. As for the Tone Generators this frequency is determined by placing a divider count in the registers. The divider value may range from 1 to 65535 with Register 11 holding the least significant eight bits and Register 12 the most significant. The master frequency for the Envelope Generator is 6991 Hz so the envelope frequency may range from 6991 Hz (divide by 1) to 0.11 Hz (divide by 65535).

Register 13

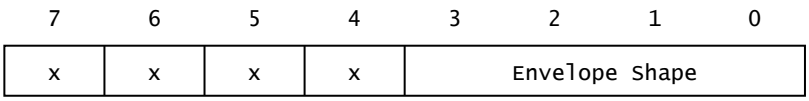


Figure 30

The four Envelope Shape bits determine the shape of the amplitude modulation envelope produced by the Envelope Generator:

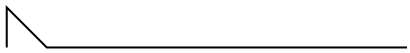
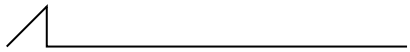
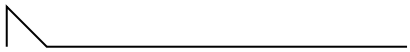
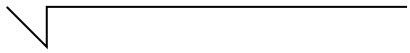
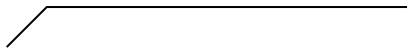
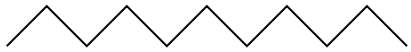
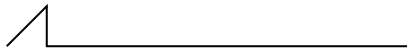
3	2	1	0	Modulation Envelope
0	0	x	x	
0	1	x	x	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

Figure 31

Register 14

7	6	5	4	3	2	1	0
Cas Input	Kbd Mode	Joy Trg.B	Joy Trg.A	Joy Right	Joy Left	Joy Back	Joy Fwd

Figure 32

This register is used to read in PSG Port A. The six joystick bits reflect the state of the four direction switches and two trigger buttons on a joystick: 0=Pressed, 1=Not pressed. Alternatively up to six Paddles may be connected instead of one joystick. Although most MSX machines have two 9 pin joystick connectors only one can be read at a time. The one to be selected for reading is determined by the Joystick Select bit in PSG Register 15.

The Keyboard Mode bit is unused on UK machines. On Japanese machines it is tied to a jumper link to determine the keyboard's character set.

The Cassette Input is used to read the signal from the cassette EAR output. This is passed through a comparator to clean the edges and to convert to digital levels but is otherwise unprocessed.

Register 15

7	6	5	4	3	2	1	0
Kana LED	Joy Sel	Pulse 2	Pulse 1	1	1	1	1

Figure 33

This register is used to output to PSG Port B. The four least significant bits are connected via TTL open-collector buffers to pins 6 and 7 of each joystick connector. They are normally set to a 1, when a paddle or joystick is connected, so that the pins can function as inputs. When a touchpad is connected they are used as handshaking outputs.

The two Pulse bits are used to generate a short positive- going pulse to any paddles attached to joystick connectors 1 or 2. Each paddle contains a monostable timer with a variable resistor controlling its pulse length. Once the timer is triggered the position of the variable resistor can be determined by counting until the monostable times out.

The Joystick Select bit determines which joystick connector is connected to PSG Port A for input: 0=Connector 1, 1=Connector 2.

The Kana LED output is unused on UK machines. On Japanese machines it is used to drive a keyboard mode indicator.

4. ROM BIOS

The design of the MSX ROM is of importance if machine code programs are to be developed efficiently and Operate reliably. Almost every program, including the BASIC Interpreter itself, will require a certain set of primitive functions to operate. These include screen and printer drivers, a keyboard decoder and other hardware related functions. By separating these routines from the BASIC Interpreter they can be made available to any application program. The section of ROM from 0000H to 268BH is largely devoted to such routines and is called the ROM BIOS (Basic Input Output System).

This chapter gives a functional description of every recognizably separate routine in the ROM BIOS. Special attention is given to the "standard" routines. These are documented by Microsoft and guaranteed to remain consistent through possible hardware and software changes. The first few

hundred bytes of the ROM consists of Z80 JP instructions which provide fixed position entry points to these routines. For maximum compatibility with future software an application program should restrict its dependence on the ROM to these locations only. The description of the ROM begins with this list of entry points to the standard routines. A brief comment is placed with each entry point, the full description is given with the routine itself.

Data Areas

It is expected that most users will wish to disassemble the ROM to some extent (the full listing runs to nearly four hundred pages). In order to ease this process the data areas, which do not contain executable Z80 code, are shown below:

```
0004H-0007H     185DH-1863H     4B3AH-4B4CH     73E4H-73E4H
002BH-002FH     1B97H-1BAAH     4C2FH-4C3FH     752EH-7585H
0508H-050DH     1BBFH-23BEH     555AH-5569H     7754H-7757H
092FH-097FH     2439H-2459H     5D83H-5DB0H     7BA3H-7BCAH
0DA5H-0EC4H     2CF1H-2E70H     6F76H-6F8EH     7ED8H-7F26H
1033H-105AH     3030H-3039H     70FFH-710CH     7F41H-7FB6H
1061H-10C1H     3710H-3719H     7182H-7195H     7FBEH-7FFFH
1233H-1252H     392EH-3FE1H     71A2H-71B5H
13A9H-1448H     43B5H-43C3H     71C7H-71DAH
160BH-1612H     46E6H-46E7H     72A6H-72B9H
```

Note that these data areas are for the UK ROM, there are slight differences in the Japanese ROM relating to the keyboard decoder and the video character set. Disparities between the ROMs are restricted to these regions with the bulk of the code being identical in both cases.

Terminology

Reference is frequently made in this chapter to the standard routines and to Workspace Area variables. Whenever this is done the Microsoft-recommended name is used in upper case letters, for example "the FILVRM standard routine" and "SCRMOD is set". Subroutines which are not named are referred to by a parenthesized address, "the screen is cleared (0777H)" for example. When reference is made to the Z80 status flags assembly language conventions are used, for example "Flag C" would mean that the carry flag is set while "Flag NZ" means that the zero flag is reset. The terms "EI" and "DI" mean enabled interrupts and disabled interrupts respectively.

ADDRESS	NAME	TO	FUNCTION
0000H	CHKRAM	02D7H	Power-up, check RAM
0004H	.....	.....	Two bytes, address of ROM character set
0006H	.....	.....	One byte, VDP Data Port number

ADDRESS	NAME	TO	FUNCTION
0007H	.....	.....	One byte, VDP Data Port number
0008H	SYNCHR	2683H	Check BASIC program character
000BH	.....	.....	NOP
000CH	RDSLT	01B6H	Read RAM in any slot
000FH	.....	.....	NOP
0010H	CHRGTR	2686H	Get next BASIC program character
0013H	.....	.....	NOP
0014H	WRSLT	01D1H	Write to RAM in any slot
0017H	.....	.....	NOP
0018H	OUTDO	1B45H	Output to current device
001BH	.....	.....	NOP
001CH	CALSLT	0217H	Call routine in any slot
001FH	.....	.....	NOP
0020H	DCOMPR	146AH	Compare register pairs HL and DE
0023H	.....	.....	NOP
0024H	ENASLT	025EH	Enable any slot permanently
0027H	.....	.....	NOP
0028H	GETYPR	2689H	Get BASIC operand type
002BH	.....	.....	Five bytes Version Number
0030H	CALLF	0205H	Call routine in any slot
0033H	.....	.....	Five NOPs
0038H	KEYINT	0C3CH	Interrupt handler, keyboard scan
003BH	INITIO	049DH	Initialize I/O devices
003EH	INIFNK	139DH	Initialize function key strings
0041H	DISSCR	0577H	Disable screen
0044H	ENASCR	0570H	Enable screen
0047H	WRTVDP	057FH	Write to any VDP register

ADDRESS	NAME	TO	FUNCTION
004AH	RDVRM	07D7H	Read byte from VRAM
004DH	WRTVRM	07CDH	Write byte to VRAM
0050H	SETRD	07ECH	Set up VDP for read
0053H	SETWRT	07DFH	Set up VDP for write
0056H	FILVRM	0815H	Fill block of VRAM with data byte
0059H	LDIRMV	070FH	Copy block to memory from VRAM
005CH	LDIRVM	0744H	Copy block to VRAM, from memory
005FH	CHGMOD	084FH	Change VDP mode
0062H	CHGCLR	07F7H	Change VDP colours
0065H	.....	.....	NOP
0066H	NMI	1398H	Non Maskable Interrupt handler
0069H	CLRSPR	06A8H	Clear all sprites
006CH	INITXT	050EH	Initialize VDP to 40x24 Text Mode
006FH	INIT32	0538H	Initialize VDP to 32x24 Text Mode
0072H	INIGRP	05D2H	Initialize VDP to Graphics Mode
0075H	INIMLT	061FH	Initialize VDP to Multicolour Mode
0078H	SETTXT	0594H	Set VDP to 40x24 Text Mode
007BH	SETT32	05B4H	Set VDP to 32x24 Text Mode
007EH	SETGRP	0602H	Set VDP to Graphics Mode
0081H	SETMLT	0659H	Set VDP to Multicolour Mode
0084H	CALPAT	06E4H	Calculate address of sprite pattern
0087H	CALATR	06F9H	Calculate address of sprite attribute
008AH	GSPSIZ	0704H	Get sprite size
008DH	GRPPRT	1510H	Print character on graphic screen
0090H	GICINI	04BDH	Initialize PSG (GI Chip)
0093H	WRTPSG	1102H	Write to any PSG register
0096H	RDPSG	110EH	Read from any PSG register

ADDRESS	NAME	TO	FUNCTION
0099H	STRTMS	11C4H	Start music dequeueing
009CH	CHSNS	0D6AH	Sense keyboard buffer for character
009FH	CHGET	10CBH	Get character from keyboard buffer (wait)
00A2H	CHPUT	08BCH	Screen character output
00A5H	LPTOUT	085DH	Line printer character output
00A8H	LPTSTT	0884H	Line printer status test
00ABH	CNVCHR	089DH	Convert character with graphic header
00AEH	PINLIN	23BFH	Get line from console (editor)
00B1H	INLIN	23D5H	Get line from console (editor)
00B4H	QINLIN	23CCH	Display "`?`", get line from console (editor)
00B7H	BREAKX	046FH	Check CTRL-STOP key directly
00BAH	ISCNTC	03FBH	Check CRTL-STOP key
00BDH	CKCNTC	10F9H	Check CTRL-STOP key
00C0H	BEEP	1113H	Go beep
00C3H	CLS	0848H	Clear screen
00C6H	POSIT	088EH	Set cursor position
00C9H	FNKSB	0B26H	Check if function key display on
00CCH	ERAFNK	0B15H	Erase function key display
00CFH	DSPFNK	0B2BH	Display function keys
00D2H	TOTEXT	083BH	Return VDP to text mode
00D5H	GTSTCK	11EEH	Get joystick status
00D8H	GTTRIG	1253H	Get trigger status
00DBH	GTPAD	12ACH	Get touch pad status
00DEH	GTPDL	1273H	Get paddle status
00E1H	TAPION	1A63H	Tape input ON
00E4H	TAPIN	1ABCH	Tape input
00E7H	TAPIOF	19E9H	Tape input OFF

ADDRESS	NAME	TO	FUNCTION
00EAH	TAPOON	19F1H	Tape output ON
00EDH	TAPOUT	1A19H	Tape output
00F0H	TAPOOF	19DDH	Tape output OFF
00F3H	STMOTR	1384H	Turn motor ON/OFF
00F6H	LFTQ	14EBH	Space left in music queue
00F9H	PUTQ	1492H	Put byte in music queue
00FCH	RIGHTC	16C5H	Move current pixel physical address right
00FFH	LEFTC	16EEH	Move current pixel physical address left
0102H	UPC	175DH	Move current pixel physical address up
0105H	TUPC	173CH	Test then UPC if legal
0108H	DOWNC	172AH	Move current pixel physical address down
010BH	TDOWNC	170AH	Test then DOWNC if legal
010EH	SCALXY	1599H	Scale graphics coordinates
0111H	MAPXYC	15DFH	Map graphic coordinates to physical address
0114H	FETCHC	1639H	Fetch current pixel physical address
0117H	STOREC	1640H	Store current pixel physical address
011AH	SETATR	1676H	Set attribute byte
011DH	READC	1647H	Read attribute of current pixel
0120H	SETC	167EH	Set attribute of current pixel
0123H	NSETCX	1809H	Set attribute of number of pixels
0126H	GTASPC	18C7H	Get aspect ratio
0129H	PNTINI	18CFH	Paint initialize
012CH	SCANR	18E4H	Scan pixels to right
012FH	SCANL	197AH	Scan pixels to left
0132H	CHGCAP	0F3DH	Change Caps Lock LED
0135H	CHGSND	0F7AH	Change Key Click sound output
0138H	RSLREG	144CH	Read Primary Slot Register

ADDRESS	NAME	TO	FUNCTION
013BH	WSLREG	144FH	Write to Primary Slot Register
013EH	RDVDP	1449H	Read VDP Status Register
0141H	SNSMAT	1452H	Read row of keyboard matrix
0144H	PHYDIO	148AH	Disk, no action
0147H	FORMAT	148EH	Disk, no action
014AH	ISFLIO	145FH	Check for file I/O
014DH	OUTDLP	1B63H	Formatted output to line printer
0150H	GETVCP	1470H	Get music voice pointer
0153H	GETVC2	1474H	Get music voice pointer
0156H	KILBUF	0468H	Clear keyboard buffer
0159H	CALBAS	01FFH	Call to BASIC from any slot
015CH	.....	.....	NOPs to 01B5H for expansion

```
Address... 01B6H
Name...... RDSLT
Entry..... A=Slot ID, HL=Address
Exit...... A=Byte read
Modifies.. AF, BC, DE, DI
```

Standard routine to read a single byte from memory in any slot. The Slot Identifier is composed of a Primary Slot number a Secondary Slot number and a flag:

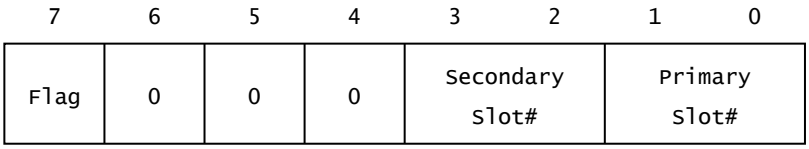


Figure 34: Slot ID

The flag is normally 0 but must be 1 if a Secondary Slot number is included in the Slot ID. The memory address and Slot ID are first processed (027EH) to yield a set of bit masks to apply to the relevant slot register. If a Secondary Slot number is specified then the Secondary Slot Register is first modified to select the relevant page from that Secondary Slot (02A3H). The Primary Slot is then switched in to the Z80 address space, the byte read and the Primary Slot restored to its original setting via the RDPRIM routine in the Workspace Area. Finally, if a Secondary Slot number is included in the Slot ID, the original Secondary Slot Register setting is restored (01ECH).

Note that, unless it is the slot containing the Workspace Area, any attempt to access page 3 (C000H to FFFFH) will cause the system to crash as RDPRIM will switch itself out. Note also that interrupts are left disabled by all the memory switching routines.

```
Address... 01D1H
Name...... WRSLT
Entry..... A=Slot ID, HL=Address, E=Byte to write
Exit...... None
Modifies.. AF, BC, D, DI
```

Standard routine to write a single byte to memory in any slot. Its operation is fundamentally the same as that of the RDSLT standard routine except that the Workspace Area routine WRPRIM is used rather than RDPRIM.

```
Address... 01FFH
Name...... CALBAS
Entry..... IX=Address
Exit...... None
Modifies.. AF', BC', DE', HL', IY, DI
```

Standard routine to call an address in the BASIC Interpreter from any slot. Usually this will be from a machine code program running in an extension ROM in page 1 (4000H to 7FFFH). The high byte of register pair IY is loaded with the MSX ROM Slot ID (00H) and control transfers to the CALSLT standard routine.

```
Address... 0205H
Name...... CALLF
Entry..... None
Exit...... None
Modifies.. AF', BC', DE', HL', IX, IY, DI
```

Standard routine to call an address in any slot. The Slot ID and address are supplied as inline parameters rather than in registers to fit inside a hook (Chapter 6), for example:

```
RST 30H
DEFB Slot ID
DEFW Address
RET
```

The Slot ID is first collected and placed in the high byte of register pair IY. The address is then placed in register pair IX and control drops into the CALSLT standard routine.

```
Address... 0217H
Name...... CALSLT
```

```
Entry..... IY(High byte)=Slot ID, IX=Address
Exit...... None
Modifies.. AF', BC', DE', HL', DI
```

Standard routine to call an address in any slot. Its operation is fundamentally the same as that of the RDSLT standard routine except that the Workspace Area routine CLPRIM is used rather than RDPRIM. Note that CALBAS and CALLF are just specialized entry points to this standard routine which offer a reduction in the amount of code required.

```
Address... 025EH
Name...... ENASLT
Entry..... A=Slot ID, HL=Address
Exit...... None
Modifies.. AF, BC, DE, DI
```

Standard routine to switch in a page permanently from any slot. Unlike the RDSLT, WRSLT and CALSLT standard routines the Primary Slot switching is performed directly and not by a Workspace Area routine. Consequently addresses in page 0 (0000H to 3FFFH) will cause an immediate system crash.

```
Address... 027EH
```

This routine is used by the memory switching standard routines to turn an address, in register pair HL, and a Slot ID, in register A, into a set of bit masks. As an example a Slot ID of FxxxSSPP and an address in Page 1 (4000H to 7FFFH) would return the following:

```
Register B=00 00 PP 00 (OR mask)
Register C=11 11 00 11 (AND mask)
Register D=PP PP PP PP (Replicated)
Register E=00 00 11 00 (Page mask)
```

Registers B and C are derived from the Primary Slot number and the page mask. They are later used to mix the new Primary Slot number into the existing contents of the Primary Slot Register. Register D contains the Primary Slot number replicated four times and register E the page mask. This is produced by examining the two most significant bits of the address, to determine the page number, and then shifting the mask along to the relevant position. These registers are later used during Secondary Slot switching.

As the routine terminates bit 7 of the Slot ID is tested, to determine whether a Secondary Slot has been specified, and Flag M returned if this is so.

```
Address... 02A3H
```

This routine is used by the memory switching standard routines to modify a Secondary Slot Register. The Slot ID is supplied in register A while registers D and E contain the bit masks shown in the previous routine.

Bits 6 and 7 of register D are first copied into the Primary Slot register. This switches in page 3 from the Primary Slot specified by the Slot ID and makes the required Secondary Slot Register available. This is then read from memory location FFFFH and the page mask, inverted, used to clear the required two bits. The Secondary Slot number is shifted to the relevant position and mixed in. Finally the new setting is placed in the Secondary Slot Register and the Primary Slot Register restored to its original setting.

```
Address... 02D7H
Name...... CHKRAM
Entry..... None
Exit...... None
Modifies.. AF, BC, DE, HL, SP
```

Standard routine to perform memory initialization at power- up. It non-destructively tests for RAM in pages 2 and 3 in all sixteen possible slots then sets the Primary and Secondary Slot registers to switch in the largest area found. The entire Workspace Area (F380H to FFC9H) is zeroed and EXPTBL and SLTTBL filled in to map any expansion interfaces in existence Interrupt Mode 1 is set and control transfers to the remainder of the power-up initialization routine (7C76H).

```
Address... 03FBH
Name...... ISCNTC
Entry..... None
Exit...... None
Modifies.. AF, EI
```

Standard routine to check whether the CTRL-STOP or STOP keys have been pressed. It is used by the BASIC Interpreter at the end of each statement to check for program termination. BASROM is first examined to see if it contains a non-zero value, if so the routine terminates immediately. This is to prevent users breaking into any extension ROM containing a BASIC program.

INTFLG is then checked to determine whether the interrupt handler has placed the CTRL-STOP or STOP key codes (03H or 04H) there. If STOP has been detected then the cursor is turned on (09DAH) and INTFLG continually checked until one of the two key codes reappears. The cursor is then turned off (0A27H) and, if the key is STOP, the routine terminates.

If CTRL-STOP has been detected then the keyboard buffer is first cleared via the KILBUF standard routine and TRPTBL is checked to see whether an "`ON STOP GOSUB`" statement is active. If so the relevant entry in TRPTBL is updated (0EF1H) and the routine terminates as the event will be handled by the Interpreter Runloop. Otherwise the ENASLT standard routine is used to switch in page 1 from the MSX ROM, in case an extension ROM is using the routine, and control transfers to the "`STOP`" statement handler (63E6H).

```
Address... 0468H
Name...... KILBUF
Entry..... None
Exit...... None
Modifies.. HL
```

Standard Routine to clear the forty character type-ahead keyboard buffer KEYBUF. There are two pointers into this buffer, PUTPNT where the interrupt handler places characters, and GETPNT where application programs fetch them from. As the number of characters in the buffer is indicated by the difference between these two pointers KEYBUF is emptied simply by making them both equal.

```
Address... 046FH
Name...... BREAKX
Entry..... None
Exit...... Flag C if CTRL-STOP key pressed
Modifies.. AF
```

Standard routine which directly tests rows 6 and 7 of the keyboard to determine whether the CTRL and STOP keys are both pressed. If they are then KEYBUF is cleared and row 7 of OLDKEY modified to prevent the interrupt handler picking the keys up as well. This routine may often be more suitable for use by an application program, in preference to ISCNTC, as it will work when interrupts are disabled, during cassette I/O for example, and does not exit to the Interpreter.

```
Address... 049DH
Name...... INITIO
Entry..... None
Exit...... None
Modifies.. AF, E, EI
```

Standard routine to initialize the PSG and the Centronics Status Port. PSG Register 7 is first set to 80H making PSG Port B=Output and PSG Port A=Input. PSG Register 15 is set to CFH to initialize the Joystick connector control hardware. PSG Register 14 is then read and the Keyboard Mode bit placed in KANAMD, this has no relevance for UK machines.

Finally a value of FFH is output to the Centronics Status Port (I/O port 90H) to set the STROBE signal high. Control then drops into the GICINI standard routine to complete initialization.

```
Address... 04BDH
Name...... GICINI
Entry..... None
Exit...... None
Modifies.. EI
```

Standard routine to initialize the PSG and the Workspace Area variables associated with the " `PLAY` " statement. QUETAB, VCBA, VCBB and VCBC are first initialized with the values shown in Chapter 6. PSG Registers 8, 9 and 10 are then set to zero amplitude and PSG Register 7 to B8H. This enables the Tone Generator and disables the Noise Generator on each channel.

```
Address... 0508H
```

This six byte table contains the " `PLAY` " statement parameters initially placed in VCBA, VCBB and VCBC by the GICINI standard routine: Octave=4, Length=4, Tempo=120, Volume=88H, Envelope=00FFH.

```
Address... 050EH
Name...... INITXT
Entry..... None
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to initialize the VDP to 40x24 Text Mode. The screen is temporarily disabled via the DISSCR standard routine and SCRMOD and OLDSCR set to 00H. The parameters required by the CHPUT standard routine are set up by copying LINL40 to LINLEN, TXTNAM to NAMBAS and TXTCGP to CGPBAS. The VDP colours are then set by the CHGCLR standard routine and the screen is cleared (077EH). The current character set is copied into the VRAM Character Pattern Table (071EH). Finally the VDP mode and base addresses are set via the SETTXT standard routine and the screen is enabled.

```
Address... 0538H
Name...... INIT32
Entry..... None
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to initialize the VDP to 32x24 Text Mode. The screen is temporarily disabled via the DISSCR standard routine and SCRMOD and OLDSCR set to 01H. The parameters required by the CHPUT standard routine are set up by copying LINL32 to LINLEN, T32NAM to NAMBAS, T32CGP to CGPBAS, T32PAT to PATBAS and T32ATR to ATRBAS. The VDP colours are then set via the CHGCLR standard routine and the screen is cleared (077EH). The current character set is copied into the VRAM Character Pattern Table (071EH) and all sprites cleared (06BBH). Finally the VDP mode and base addresses are set via the SETT32 standard routine and the screen is enabled.

```
Address... 0570H
Name...... ENASCR
Entry..... None
Exit...... None
Modifies.. AF, BC, EI
```

Standard routine to enable the screen. This simply involves setting bit 6 of VDP Mode Register 1.

```
Address... 0577H
Name...... DISSCR
Entry..... None
Exit...... None
Modifies.. AF, BC, EI
```

Standard routine to disable the screen. This simply involves resetting bit 6 of VDP Mode Register 1.

```
Address... 057FH
Name...... WRTVDP
Entry..... B=Data byte, C=VDP Mode Register number
Exit...... None
Modifies.. AF, B, EI
```

Standard routine to write a data byte to any VDP Mode Register. The register selection byte is first written to the VDP Command Port, followed by the data byte. This is then copied to the relevant register image, RG0SAV to RG7SAV, in the Workspace Area

```
Address... 0594H
Name...... SETTXT
Entry..... None
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to partially set the VDP to 40x24 Text Mode. The mode bits M1, M2 and M3 are first set in VDP Mode Registers 0 and 1. The five VRAM table base addresses, beginning with TXTNAM, are then copied from the Workspace Area into VDP Mode Registers 2, 3, 4, 5 and 6 (0677H).

```
Address... 05B4H
Name...... SETT32
Entry..... None
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to partially set the VDP to 32x24 Text Mode. The mode bits M1, M2 and M3 are first set in VDP Mode Registers 0 and 1. The five VRAM table base addresses, beginning with T32NAM, are then copied from the Workspace Area into VDP Mode Registers 2, 3, 4, 5 and 6 (0677H).

```
Address... 05D2H
Name...... INIGRP
Entry..... None
```

```
Exit..... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to initialize the VDP to Graphics Mode. The screen is temporarily disabled via the DISSCR standard routine and SCRMOD set to 02H. The parameters required by the GRPPRT standard routine are set up by copying GRPPAT to PATBAS and GRPATR to ATRBAS. The character code driver pattern is then copied into the VDP Name Table, the screen cleared (07A1H) and all sprites cleared (06BBH). Finally the VDP mode and base addresses are set via the SETGRP standard routine and the screen is enabled.

```
Address... 0602H
Name..... SETGRP
Entry..... None
Exit..... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to partially set the VDP to Graphics Mode. The mode bits M1, M2 and M3 are first set in VDP Mode Registers 0 and 1. The five VRAM table base addresses, beginning with GRPNAM, are then copied from the Workspace Area into VDP Mode Registers 2, 3, 4, 5 and 6 (0677H).

```
Address... 061FH
Name..... INIMLT
Entry..... None
Exit..... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to initialize the VDP to Multicolour Mode. The screen is temporarily disabled via the DISSCR standard routine and SCRMOD set to 03H. The parameters required by the GRPPRT standard routine are set up by copying MLTPAT to PATBAS and MLTATR to ATRBAS. The character code driver pattern is then copied into the VDP Name Table, the screen cleared (07B9H) and all sprites cleared (06BBH). Finally the VDP mode and base addresses are set via the SETMLT standard routine and the screen is enabled.

```
Address... 0659H
Name..... SETMLT
Entry..... None
Exit..... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to partially set the VDP to Multicolour Mode. The mode bits M1, M2 and M3 are first set in VDP Mode Registers 0 and 1. The five VRAM table base addresses, beginning with MLTNAM, are then copied from the Workspace Area to VDP Mode Registers 2, 3, 4, 5 and 6.

```
Address... 0677H
```

This routine is used by the SETTXT, SETT32, SETGRP and SETMLT standard routines to copy a block of five table base addresses from the Workspace Area into VDP Mode Registers 2, 3, 4, 5 and 6. On entry register pair HL points to the relevant group of addresses. Each base address is collected in turn shifted the required number of places and then written to the relevant Mode Register via the WRTVDP standard routine.

```
Address... 06A8H
Name..... CLRSPR
Entry..... None
Exit..... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to clear all sprites. The entire 2 KB Sprite Pattern Table is first filled with zeros via the FILVRM standard routine. The vertical coordinate of each of the thirty-two sprite attribute blocks is then set to -47 (D1H) to place the sprite above the top of the screen, the horizontal coordinate is left unchanged.

The pattern numbers in the Sprite Attribute Table are initialized with the series 0, 1, 2, 3, 4,... 31 for 8x8 sprites or the series 0, 4, 8, 12, 16,... 124 for 16x16 sprites. The series to be generated is determined by the Size bit in VDP Mode Register 1. Finally the colour byte of each sprite attribute block is filled in with the colour code contained in FORCLR, this is initially white.

Note that the Size and Mag bits in VDP Mode Register 1 are not affected by this routine. Note also that the INIT32, INIGRP and INIMLT standard routines use this routine with an entry point at 06BBH, leaving the Sprite Pattern Table undisturbed.

```
Address... 06E4H
Name..... CALPAT
Entry..... A=Sprite pattern number
Exit..... HL=Sprite pattern address
Modifies.. AF, DE, HL
```

Standard routine to calculate the address of a sprite pattern. The pattern number is first multiplied by eight then, if 16x16 sprites are selected, multiplied by a further factor of four. This is then added to the Sprite Pattern Table base address, taken from PATBAS, to produce the final address.

This numbering system is in line with the BASIC Interpreter's usage of pattern numbers rather than the VDP's when 16x16 sprites are selected. As an example while the Interpreter calls the second pattern number one, it is actually VDP pattern number four. This usage means that the maximum pattern number this routine should allow, when 16x16 sprites are selected, is sixty-three. There is no actual check on this limit so large pattern numbers will produce addresses greater than 3FFFH. Such

addresses, when passed to the other VDP routines, will wrap around past zero and corrupt the Character Pattern Table in VRAM.

```
Address... 06F9H
Name..... CALATR
Entry..... A=Sprite number
Exit..... HL=Sprite attribute address
Modifies.. AF, DE, HL
```

Standard routine to calculate the address of a sprite attribute block. The sprite number, from zero to thirty-one, is multiplied by four and added to the Sprite Attribute Table base address taken from ATRBAS.

```
Address... 0704H
Name..... GSPSIZ
Entry..... None
Exit..... A=Bytes in sprite pattern (8 or 32)
Modifies.. AF
```

Standard routine to return the number of bytes occupied by each sprite pattern in the Sprite Pattern Table. The result is determined simply by examining the Size bit in VDP Mode Register 1.

```
Address... 070FH
Name..... LDIRMV
Entry..... BC=Length, DE=RAM address, HL=VRAM address
Exit..... None
Modifies.. AF, BC, DE, EI
```

Standard routine to copy a block into main memory from the VDP VRAM. The VRAM starting address is set via the SETRD standard routine and then sequential bytes read from the VDP Data Port and placed in main memory.

```
Address... 071EH
```

This routine is used to copy a 2 KB character set into the VDP Character Pattern Table in any mode. The base address of the Character Pattern Table in VRAM is taken from CGPBAS. The starting address of the character set is taken from CGPNT. The RDSLT standard routine is used to read the character data so this may be situated in an extension ROM.

At power-up CGPNT is initialized with the address contained at ROM location 0004H, which is 1BBFH. CGPNT is easily altered to produce some interesting results, `POKE &HF920,&HC7:SCREEN 0` provides a thoroughly confusing example.

```
Address... 0744H
Name...... LDIRVM
Entry..... BC=Length, DE=VRAM address, HL=RAM address
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to copy a block to VRAM from main memory. The VRAM starting address is set via the SETWRT standard routine and then sequential bytes taken from main memory and written to the VDP Data Port.

```
Address... 0777H
```

This routine will clear the screen in any VDP mode. In 40x24 Text Mode and 32x24 Text Mode the Name Table, whose base address is taken from NAMBAS, is first filled with ASCII spaces. The cursor is then set to the home position (0A7FH) and LINTTB, the line termination table, re-initialized. Finally the function key display is restored, if it is enabled, via the FNKSB standard routine.

In Graphics Mode the border colour is first set via VDP Mode Register 7 (0832H). The Colour Table is then filled with the background colour code, taken from BAKCLR, for both 0 and 1 pixels. Finally the Character Pattern Table is filled with zeroes.

In Multicolour Mode the border colour is first set via VDP Mode Register 7 (0832H). The Character Pattern Table is then filled with the background colour taken from BAKCLR.

```
Address... 07CDH
Name...... WRTVRM
Entry..... A=Data byte, HL=VRAM address
Exit...... None
Modifies.. EI
```

Standard routine to write a single byte to the VDP VRAM. The VRAM address is first set up via the SETWRT standard routine and then the data byte written to the VDP Data Port. Note that the two seemingly spurious `EX(SP),HL` instructions in this routine, and several others, are required to meet the VDP's timing constraints.

```
Address... 07D7H
Name...... RDVRM
Entry..... HL=VRAM address
Exit...... A=Byte read
Modifies.. AF, EI
```

Standard routine to read a single byte from the VDP VRAM. The VRAM address is first set up via the SETRD standard routine and then the byte read from the VDP Data Port.

```
Address... 07DFH
Name...... SETWRT
Entry..... HL=VRAM address
Exit...... None
Modifies.. AF, EI
```

Standard routine to set up the VDP for subsequent writes to VRAM via the Data Port. The address contained in register pair HL is written to the VDP Command Port LSB first, MSB second as shown in Figure 7. Addresses greater than 3FFFH will wrap around past zero as the two most significant bits of the address are ignored.

```
Address... 07ECH
Name...... SETRD
Entry..... HL=VRAM address
Exit...... None
Modifies.. AF, EI
```

Standard routine to set up the VDP for subsequent reads from VRAM via the Data Port. The address contained in register pair HL is written to the VDP Command Port LSB first, MSB second as shown in Figure 7. Addresses greater than 3FFFH will wrap around past zero as the two most significant bits of the address are ignored.

```
Address... 07F7H
Name...... CHGCLR
Entry..... None
Exit...... None
Modifies.. AF, BC, HL, EI
```

Standard routine to set the VDP colours. SCRMOD is first examined to determine the appropriate course of action. In 40x24 Text Mode the contents of BAKCLR and FORCLR are written to VDP Mode Register 7 to set the colour of the 0 and 1 pixels, these are initially blue and white. Note that in this mode there is no way of specifying the border colour, this will be the same as the 0 pixel colour. In 32x24 Text Mode, Graphics Mode or Multicolour Mode the contents of BDRCLR are written to VDP Mode Register 7 to set the colour of the border, this is initially blue. Also in 32x24 Text Mode the contents of BAKCLR and FORCLR are copied to the whole of the Colour Table to determine the 0 and 1 pixel colours.

```
Address... 0815H
Name...... FILVRM
Entry..... A=Data byte, BC=Length, HL=VRAM address
Exit...... None
Modifies.. AF, BC, EI
```

Standard routine to fill a block of the VDP VRAM with a single data byte. The VRAM starting address, contained in register pair HL, is first set up via the SETWRT standard routine. The data byte is then repeatedly written to the VDP Data Port to fill successive VRAM locations.

```
Address... 083BH
Name...... TOTEXT
Entry..... None
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to return the VDP to either 40x24 Text Mode or 32x24 Text Mode if it is currently in Graphics Mode or Multicolour Mode. It is used by the BASIC Interpreter Mainloop and by the "INPUT" statement handler. Whenever the INITXT or INIT32 standard routines are used the mode byte, 00H or 01H, is copied into OLDSCR. If the mode is subsequently changed to Graphics Mode or Multicolour Mode, and then has to be returned to one of the two text modes for keyboard input, this routine ensures that it returns to the same one.

SCRMOD is first examined and, if the screen is already in either text mode, the routine simply terminates with no action. Otherwise the previous text mode is taken from OLDSCR and passed to the CHGMOD standard routine.

```
Address... 0848H
Name...... CLS
Entry..... Flag Z
Exit...... None
Modifies.. AF, BC, DE, EI
```

Standard routine to clear the screen in any mode, it does nothing but call the routine at 0777H. This is actually the "`CLS`" statement handler and, because this indicates that there is illegal text after the statement, it will simply return if entered with Flag NZ.

```
Address... 084FH
Name...... CHGMOD
Entry..... A=Screen mode required (0, 1, 2, 3)
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to set a new screen mode. Register A, containing the required screen mode, is tested and control transferred to INITXT, INIT32, INIGRP or INIMLT.

```
Address... 085DH
Name...... LPTOUT
Entry..... A=Character to print
```

```
Exit..... Flag C if CTRL-STOP termination
Modifies.. AF
```

Standard routine to output a character to the line printer via the Centronics Port. The printer status is continually tested, via the LPTSTT standard routine, until the printer becomes free. The character is then written to the Centronics Data Port (I/O port 91H) and the STROBE signal of the Centronics Status Port (I/O port 90H) briefly pulsed low. Note that the BREAKX standard routine is used to test for the CTRL-STOP key if the printer is busy. If CTRL-STOP is detected a CR code is written to the Centronics Data Port, to flush the printer's line buffer, and the routine terminates with Flag C.

```
Address... 0884H
Name..... LPTSTT
Entry..... None
Exit..... A=0 and Flag Z if printer busy
Modifies.. AF
```

Standard routine to test the Centronics Status Port BUSY signal. This just involves reading I/O port 90H and examining the state of bit 1: 0=Ready, 1=Busy.

```
Address... 088EH
Name..... POSIT
Entry..... H=Column, L=Row
Exit..... None
Modifies.. AF, EI
```

Standard routine to set the cursor coordinates. The row and column coordinates are sent to the OUTDO standard routine as the parameters in an ESC,"Y",Row+1FH, Column+1FH sequence. Note that the BIOS home position has coordinates of 1,1 rather than the 0,0 used by the BASIC Interpreter.

```
Address... 089DH
Name..... CNVCHR
Entry..... A=Character
Exit..... Flag Z,NC=Header; Flag NZ,C=Graphic; Flag Z,C=Normal
Modifies.. AF
```

Standard routine to test for, and convert if necessary, characters with graphic headers. Characters less than 20H are normally interpreted by the output device drivers as control characters. A character code in this range can be treated as a displayable character by preceding it with a graphic header control code (01H) and adding 40H to its value. For example to directly display character code 0DH, rather than have it interpreted as a carriage return, it is necessary to output the two bytes 01H,4DH. This routine is used by the output device drivers, such as the CHPUT standard routine, to check for such sequences.

If the character is a graphic header GRPHED is set to 01H and the routine terminates, otherwise GRPHED is zeroed. If the character is outside the range 40H to 5FH it is left unchanged. If it is inside this range, and GRPHED contains 01H indicating a previous graphic header, it is converted by subtracting 40H.

```
Address... 08BCH
Name...... CHPUT
Entry..... A=Character
Exit...... None
Modifies.. EI
```

Standard routine to output a character to the screen in 40x24 Text Mode or 32x24 Text Mode. SCRMOD is first checked and, if the VDP is in either Graphics Mode or Multicolour Mode, the routine terminates with no action. Otherwise the cursor is removed (0A2EH), the character decoded (08DFH) and then the cursor replaced (09E1H). Finally the cursor column position is placed in TTYPOS, for use by the "`PRINT`" statement, and the routine terminates.

```
Address... 08DFH
```

This routine is used by the CHPUT standard routine to decode a character and take the appropriate action. The CNVCHR standard routine is first used to check for a graphic character, if the character is a header code (01H) then the routine terminates with no action. If the character is a converted graphic one then the control code decoding section is skipped. Otherwise ESCCNT is checked to see if a previous ESC character (1BH) has been received, if so control transfers to the ESC sequence processor (098FH). Otherwise the character is checked to see if it is smaller than 20H, if so control transfers to the control code processor (0914H). The character is then checked to see if it is DEL (7FH), if so control transfers to the delete routine (0AE3H).

Assuming the character is displayable the cursor coordinates are taken from CSRY and CSRX and placed in register pair HL, H=Column, L=Row. These are then converted to a physical address in the VDP Name Table and the character placed there (0BE6H). The cursor column position is then incremented (0A44H) and, assuming the rightmost column has not been exceeded, the routine terminates. Otherwise the row's entry in LINTTB, the line termination table, is zeroed to indicate an extended logical line, the column number is set to 01H and a LF is performed.

```
Address... 0908H
```

This routine performs the LF operation for the CHPUT standard routine control code processor. The cursor row is incremented (0A61H) and, assuming the lowest row has not been exceeded, the routine terminates. Otherwise the screen is scrolled upwards and the lowest row erased (0A88H).

```
Address... 0914H
```

This is the control code processor for the CHPUT standard routine. The table at 092FH is searched for a match with the code and control transferred to the associated address.

```
Address... 092FH
```

This table contains the control codes, each with an associated address, recognized by the CHPUT standard routine:

CODE	TO	FUNCTION
07H	1113H	BELL, go beep
08H	0A4CH	BS, cursor left
09H	0A71H	TAB, cursor to next tab position
0AH	0908H	LF, cursor down a row
0BH	0A7FH	HOME, cursor to home
0CH	077EH	FORMFEED, clear screen and home
0DH	0A81H	CR, cursor to leftmost column
1BH	0989H	ESC, enter escape sequence
1CH	0A5BH	RIGHT, cursor right
1DH	0A4CH	LEFT, cursor left
1EH	0A57H	UP, cursor up
1FH	0A61H	DOWN, cursor down.

```
Address... 0953H
```

This table contains the ESC control codes, each with an associated address, recognized by the CHPUT standard routine:

CODE	TO	FUNCTION
6AH	077EH	ESC,"j", clear screen and home
45H	077EH	ESC,"E", clear screen and home
4BH	0AEEH	ESC,"K", clear to end of line
4AH	0B05H	ESC,"J", clear to end of screen

CODE	TO	FUNCTION
6CH	0AECH	ESC,"l", clear line
4CH	0AB4H	ESC,"L", insert line
4DH	0A85H	ESC,"M", delete line
59H	0986H	ESC,"Y", set cursor coordinates
41H	0A57H	ESC,"A", cursor up
42H	0A61H	ESC,"B", cursor down
43H	0A44H	ESC,"C", cursor right
44H	0A55H	ESC,"D", cursor left
48H	0A7FH	ESC,"H", cursor home
78H	0980H	ESC,"x", change cursor
79H	0983H	ESC,"y", change cursor

```
Address... 0980H
```

This routine performs the ESC,"x" operation for the CHPUT standard routine control code processor. ESCCNT is set to 01H to indicate that the next character received is a parameter.

```
Address... 0983H
```

This routine performs the ESC,"y" operation for the CHPUT standard routine control code decoder. ESCCNT is set to 02H to indicate that the next character received is a parameter.

```
Address... 0986H
```

This routine performs the ESC",Y" operation for the CHPUT standard routine control code processor. ESCCNT is set to 04H to indicate that the next character received is a parameter.

```
Address... 0989H
```

This routine performs the ESC operation for the CHPUT standard routine control code processor. ESCCNT is set to FFH to indicate that the next character received is the second control character.

```
Address... 098FH
```

This is the CHPUT standard routine ESC sequence processor. If ESCCNT contains FFH then the character is the second control character and control transfers to the control code processor (0919H) to search the ESC code table at 0953H.

If ESCCNT contains 01H then the character is the single parameter of the ESC,"x" sequence. If the parameter is "4" (34H) then CSTYLE is set to 00H resulting in a block cursor. If the parameter is "5" (35H) then CSRSW is set to 00H making the cursor normally disabled.

If ESCCNT contains 02H then the character is the single parameter in the ESC,"y" sequence. If the parameter is "4" (34H) then CSTYLE is set to 01H resulting in an underline cursor. If the parameter is "5" (35H) then CSRSW is set to 01H making the cursor normally enabled.

If ESCCNT contains 04H then the character is the first parameter of the ESC,"Y" sequence and is the row coordinate. The parameter has 1FH subtracted and is placed in CSRY, ESCCNT is then decremented to 03H.

If ESCCNT contains 03H then the character is the second parameter of the ESC,"Y" sequence and is the column coordinate. The parameter has 1FH subtracted and is placed in CSRX.

```
Address... 09DAH
```

This routine is used, by the CHGET standard routine for example, to display the cursor character when it is normally disabled. If CSRSW is non-zero the routine simply terminates with no action, otherwise the cursor is displayed (09E6H).

```
Address... 09E1H
```

This routine is used, by the CHPUT standard routine for example, to display the cursor character when it is normally enabled. If CSRSW is zero the routine simply terminates with no action. SCRMOD is checked and, if the screen is in Graphics Mode or Multicolour Mode, the routine terminates with no action. Otherwise the cursor coordinates are converted to a physical address in the VDP Name Table and the character read from that location (0BD8H) and saved in CURSAV.

The character's eight byte pixel pattern is read from the VDP Character Pattern Table into the LINWRK buffer (0BA5H). The pixel pattern is then inverted, all eight bytes if CSTYLE indicates a block cursor, only the bottom three if CSTYLE indicates an underline cursor. The pixel pattern is copied back to the position for character code 255 in the VDP Character Pattern Table (0BBEH). The character code 255 is then placed at the current cursor location in the VDP Name Table (0BE6H) and the routine terminates.

This method of generating the cursor character, by using character code 255, can produce curious effects under certain conditions. These can be demonstrated by executing the BASIC statement `FOR N=1 TO 100: PRINT CHR$(255);:NEXT` and then pressing the cursor up key.

```
Address... 0A27H
```

This routine is used, by the CHGET standard routine for example, to remove the cursor character when it is normally disabled. If CSRSW is non-zero the routine simply terminates with no action, otherwise the cursor is removed (0A33H).

```
Address... 0A2EH
```

This routine is used, by the CHPUT standard routine for example, .to remove the cursor character when it is normally enabled. If CSRSW is zero the routine simply terminates with no action. SCRMOD is checked and, if the screen is in Graphics Mode or Multicolour Mode, the routine terminates with no action. Otherwise the cursor coordinates are converted to a physical address in the VDP Name Table and the character held in CURSAV written to that location (0BE6H).

```
Address... 0A44H
```

This routine performs the ESC,"C" operation for the CHPUT standard routine control code processor. If the cursor column coordinate is already at the rightmost column, determined by LINLEN, then the routine terminates with no action. Otherwise the column coordinate is incremented and CSRX updated.

```
Address... 0A4CH
```

This routine performs the BS/LEFT operation for the CHPUT standard routine control code processor. The cursor column coordinate is decremented and CSRX updated. If the column coordinate has moved beyond the leftmost position it is set to the rightmost position, from LINLEN, and an UP operation performed.

```
Address... 0A55H
```

This routine performs the ESC,"D" operation for the CHPUT standard routine control code processor. If the cursor column coordinate is already at the leftmost position then the routine terminates with no action. Otherwise the column coordinate is decremented and CSRX updated.

```
Address... 0A57H
```

This routine performs the ESC,"A" (UP) operation for the CHPUT standard routine control code processor. If the cursor row coordinate is already at the topmost position the routine terminates with no action. Otherwise the row coordinate is decremented and CSRY updated.

```
Address... 0A5BH
```

This routine performs the RIGHT operation for the CHPUT standard routine control code processor. The cursor column coordinate is incremented and CSRX updated. If the column coordinate has moved beyond the rightmost position, determined by LINLEN, it is set to the leftmost position (01H) and a DOWN operation performed.

```
Address... 0A61H
```

This routine performs the ESC,"B" (DOWN) operation for the CHPUT standard routine control code processor. If the cursor row coordinate is already at the lowest position, determined by CRTCNT and CNSDFG (0C32H), then the routine terminates with no action. Otherwise the row coordinate is incremented and CSRY updated.

```
Address... 0A71H
```

This routine performs the TAB operation for the CHPUT standard routine control code processor. ASCII spaces are output (08DFH) until CSRX is a multiple of eight plus one (BIOS columns 1, 9, 17, 25, 33).

```
Address... 0A7FH
```

This routine performs the ESC,"H" (HOME) operation for the CHPUT standard routine control code processor, CSRX and CSRY are simply set to 1,1. The ROM BIOS cursor coordinate system, while functionally identical to that used by the BASIC Interpreter, numbers the screen rows from 1 to 24 and the columns from 1 to 32/40.

```
Address... 0A81H
```

This routine performs the CR operation for the CHPUT standard routine control code processor, CSRX is simply set to 01H .

```
Address... 0A85H
```

This routine performs the ESC,"M" function for the CHPUT standard routine control code processor. A CR operation is first performed to set the cursor column coordinate to the leftmost position. The number of rows from the current row to the bottom of the screen is then determined, if this is zero the current row is simply erased (0AECH). The row count is first used to scroll up the relevant section of LINTTB, the line termination table, by one byte. It is then used to scroll up the relevant section of the screen a row at a time. Starting at the row below the current row, each line is copied from the

VDP Name Table into the LINWRK buffer (0BAAH) then copied back to the Name Table one row higher (0BC3H). Finally the lowest row on the screen is erased (0AECH).

```
Address... 0AB4H
```

This routine performs the ESC,"L" operation for the CHPUT standard routine control code processor. A CR operation is first performed to set the cursor column coordinate to the leftmost position. The number of rows from the current row to the bottom of the screen is then determined, if this is zero the current row is simply erased (0AECH). The row count is first used to scroll down the relevant section of LINTTB, the line termination table, by one byte. It is then used to scroll down the relevant section of the screen a row at a time. Starting at the next to last row of the screen, each line is copied from the VDP Name Table into the LINWRK buffer (0BAAH), then copied back to the Name Table one row lower (0BC3H). Finally the current row is erased (0AECH).

```
Address... 0AE3H
```

This routine is used to perform the DEL operation for the CHPUT standard routine control code processor. A LEFT operation is first performed. If this cannot be completed, because the cursor is already at the home position, then the routine terminates with no action. Otherwise a space is written to the VDP Name Table at the cursor's physical location (0BE6H).

```
Address... 0AECH
```

This routine performs the ESC,"I" operation for the CHPUT standard routine control code processor. The cursor column coordinate is set to 01H and control drops into the ESC,"K" routine.

```
Address... 0AEEH
```

This routine performs the ESC,"K" operation for the CHPUT standard routine control code processor. The row's entry in LINTTB, the line termination table, is first made non-zero to indicate that the logical line is not extended (0C29H). The cursor coordinates are converted to a physical address (0BF2H) in the VDP Name Table and the VDP set up for writes via the SETWRT standard routine. Spaces are then written directly to the VDP Data Port until the rightmost column, determined by LINLEN, is reached.

```
Address... 0B05H
```

This routine performs the ESC,"J" operation for the CHPUT standard routine control code processor. An ESC,"K" operation is performed on successive rows, starting with the current one, until the bottom of the screen is reached.

```
Address... 0B15H
Name..... ERAFNK
Entry..... None
Exit..... None
Modifies.. AF, DE, EI
```

Standard routine to turn the function key display off. CNSDFG is first zeroed and, if the VDP is in Graphics Mode or Multicolour Mode, the routine terminates with no further action. If the VDP is in 40x24 Text Mode or 32x24 Text Mode the last row on the screen is then erased (0AECH).

```
Address... 0B26H
Name..... FNKSB
Entry..... None
Exit..... None
Modifies.. AF, BC, DE, EI
```

Standard routine to show the function key display if it is enabled. If CNSDFG is zero the routine terminates with no action, otherwise control drops into the DSPFNK standard routine..

```
Address... 0B2BH
Name..... DSPFNK
Entry..... None
Exit..... None
Modifies.. AF, BC, DE, EI
```

Standard routine to turn the function key display on. CNSDFG is set to FFH and, if the VDP is in Graphics Mode or Multicolour Mode, the routine terminates with no further action. Otherwise the cursor row coordinate is checked and, if the cursor is on the last row of the screen, a LF code (0AH) issued to the OUTDO standard routine to scroll the screen up.

Register pair HL is then set to point to either the unshifted or shifted function strings in the Workspace Area depending upon whether the SHIFT key is pressed. LINLEN has four subtracted, to allow a minimum of one space between fields, and is divided by five to determine the field size for each string. Successive characters are then taken from each function string, checked for graphic headers via the CNVCHR standard routine and placed in the LINWRK buffer until the string is exhausted or the zone is filled. When all five strings are completed the LINWRK buffer is written to the last row in the VDP Name Table (0BC3H).

```
Address... 0B9CH
```

This routine is used by the function key display related standard routines. The contents of register A are placed in CNSDFG then SCRMOD tested and Flag NC returned if the screen is in Graphics Mode or Multicolour Mode.

```
Address... 0BA5H
```

This routine copies eight bytes from the VDP VRAM into the LINWRK buffer, the VRAM physical address is supplied in register pair HL.

```
Address... 0BAAH
```

This routine copies a complete row of characters, with the length determined by LINLEN, from the VDP VRAM into the LINWRK buffer. The cursor row coordinate is supplied in register L.

```
Address... 0BBEH
```

This routine copies eight bytes from the LINWRK buffer into the VDP VRAM, the VRAM physical address is supplied in register pair HL.

```
Address... 0BC3H
```

This routine copies a complete row of characters, with the length determined by LINLEN, from the LINWRK buffer into the VDP VRAM. The cursor row coordinate is supplied in register L.

```
Address... 0BD8H
```

This routine reads a single byte from the VDP VRAM into register C. The column coordinate is supplied in register H, the row coordinate in register L.

```
Address... 0BE6H
```

This routine converts a pair of screen coordinates, the column in register H and the row in register L, into a physical address in the VDP Name Table. This address is returned in register pair HL.

The row coordinate is first multiplied by thirty-two or forty, depending upon the screen mode, and added to the column coordinate. This is then added to the Name Table base address, taken from NAMBAS, to produce an initial address.

Because of the variable screen width, as contained in LINLEN, an additional offset has to be added to the initial address to keep the active region roughly centered within the screen. The difference between the "true" number of characters per row, thirty-two or forty, and the current width is halved and then rounded up to produce the left hand offset. For a UK machine, with a thirty-seven character width in 40x24 Text Mode, this will result in two unused characters on the left hand side and one on

the right. The statement `PRINT (41-WID)\2`, where `WID` is any screen width, will display the left hand column offset in 40x24 Text Mode.

A complete BASIC program which emulates this routine is given below:

```
10 CPR=40:NAM=BASE(0):WID=PEEK(&HF3AE)
20 SCRMD=PEEK(&HFCAF):IF SCRMD=0 THEN 40
30 CPR=32:NAM=BASE(5):WID=PEEK(&HF3AF)
40 LH=(CPR+1-WID)\2
50 ADDR=NAM+(ROW-1)*CPR+(COL-1)+LH
```

This program is designed for the `ROW` and `COL` coordinate system used by the ROM BIOS where home is 1,1. Line 50 may be simplified, by removing the "-1" factors, if the BASIC Interpreter's coordinate system is to be used.

```
Address... 0C1DH
```

This routine calculates the address of a row's entry in LINTTB, the line termination table. The row coordinate is supplied in register L and the address returned in register pair DE.

```
Address... 0C29H
```

This routine makes a row's entry in LINTTB non-zero when entered at 0C29H and zero when entered at 0C2AH. The row coordinate is supplied in register L.

```
Address... 0C32H
```

This routine returns the number of rows on the screen in register A. It will normally return twenty-four if the function key display is disabled and twenty-three if it is enabled. Note that the screen size is determined by CRTCNT and may be modified with a BASIC statement, `POKE &HF3B1H,14:SCREEN 0` for example.

```
Address... 0C3CH
Name...... KEYINT
Entry..... None
Exit...... None
Modifies.. EI
```

Standard routine to process Z80 interrupts, these are generated by the VDP once every 20 ms on a UK machine. The VDP Status Register is first read and bit 7 checked to ensure that this is a frame rate interrupt, if not the routine terminates with no action. The contents of the Status Register are saved in

STATFL and bit 5 checked for sprite coincidence. If the Coincidence Flag is active then the relevant entry in TRPTBL is updated (0EF1H).

INTCNT, the "`INTERVAL`" counter, is then decremented. If this has reached zero the relevant entry in TRPTBL is updated (0EF1H) and the counter reset with the contents of INTVAL.

JIFFY, the "`TIME`" counter, is then incremented. This counter just wraps around to zero when it overflows.

MUSICF is examined to determine whether any of the three music queues generated by the "`PLAY`" statement are active. For each active queue the dequeueing routine (113BH) is called to fetch the next music packet and write it to the PSG.

SCNCNT is then decremented to determine if a joystick and keyboard scan is required, if not the interrupt handler terminates with no further action. This counter is used to increase throughput and to minimize keybounce problems by ensuring that a scan is only carried out every three interrupts. Assuming a scan is required joystick connector 1 is selected and the two Trigger bits read (120CH), followed by the two Trigger bits from joystick connector 2 (120CH) and the SPACE key from row 8 of the keyboard (1226H). These five inputs, which are all related to the "`STRIG`" statement, are combined into a single byte where 0=Pressed, 1=Not pressed:

7	6	5	4	3	2	1	0
Joy 2 Trg.B	Joy 2 Trg.A	Joy 1 Trg.B	Joy 1 Trg.A	0	0	0	Space

Figure 35: "`STRIG`" Inputs

This reading is compared with the previous one, held in TRGFLG, to produce an active transition byte and TRGFLG is updated with the new reading. The active transition byte is normally zero but contains a 1 in each position where a transition from unpressed to pressed has occurred. This active transition byte is shifted out bit by bit and the relevant entry in TRPTBL updated (0EF1H) for each active device.

A complete scan of the keyboard matrix is then performed to identify new key depressions, any found are translated into key codes and placed in KEYBUF (0D12H). If KEYBUF is found to be empty at the end of this process REPCNT is decremented to see whether the auto-repeat delay has expired, if not the routine terminates. If the delay period has expired REPCNT is reset with the fast repeat value (60 ms), the OLDKEY keyboard map is reinitialized and the keyboard scanned again (0D4EH). Any keys which are continuously pressed will show up as new transitions during this scan. Note that keys will only auto-repeat while an application program keeps KEYBUF empty by reading characters. The interrupt handler then terminates.

```
Address... 0D12H
```

This routine performs a complete scan of all eleven rows of the keyboard matrix for the interrupt handler. Each of the eleven rows is read in via the PPI and placed in ascending order in NEWKEY. ENSTOP is then checked to see if warm starts are enabled. If its contents are non-zero and the keys CODE, GRAPH, CTRL and SHIFT are pressed control transfers to the BASIC Interpreter (409BH) via the CALBAS standard routine. This facility is useful as even a machine code program can be terminated as long as the interrupt handler is running. The contents of NEWKEY are compared with the previous scan contained in OLDKEY. If any change at all has occurred REPCNT is loaded with the initial auto-repeat delay (780 ms). Each row 1, reading from NEWKEY is then compared with the previous one, held in OLDKEY, to produce an active transition byte and OLDKEY is updated with the new reading. The active transition byte is normally zero but contains a 1 in each position where a transition from unpressed to pressed has occurred. If the row contains any transitions these are decoded and placed in KEYBUF as key codes (0D89H). When all eleven rows have been completed the routine checks whether there are any characters in KEYBUF, by subtracting GETPNT from PUTPNT, and terminates.

```
Address... 0D6AH
Name...... CHSNS
Entry..... None
Exit...... Flag NZ if characters in KEYBUF
Modifies.. AF, EI
```

Standard routine to check if any keyboard characters are ready. If the screen is in Graphics Mode or Multicolour Mode then GETPNT is subtracted from PUTPNT (0D62H) and the routine terminates. If the screen is in 40x24 Text Mode or 32x24 Text Mode the state of the SHIFT key is also examined and the function key display updated, via the DSPFNK standard routine, if it has changed.

```
Address... 0D89H
```

This routine converts each active bit in a keyboard row transition byte into a key code. A bit is first converted into a key number determined by its position in the keyboard matrix:

7 (07H)	6 (06H)	5 (05H)	4 (04H)	3 (03H)	2 (02H)	1 (01H)	0 (00H)	Row 0
; (0FH)	] (0EH)	[(0DH)	\ (0CH)	= (0BH)	- (0AH)	9 (09H)	8 (08H)	Row 1
B (17H)	A (16H)	£ (15H)	/ (14H)	. (13H)	, (12H)	` (11H)	' (10H)	Row 2
J (1FH)	I (1EH)	H (1DH)	G (1CH)	F (1BH)	E (1AH)	D (19H)	C (18H)	Row 3
R (27H)	Q (26H)	P (25H)	O (24H)	N (23H)	M (22H)	L (21H)	K (20H)	Row 4
Z (2FH)	Y (2EH)	X (2DH)	W (2CH)	V (2BH)	U (2AH)	T (29H)	S (28H)	Row 5
F3 (37H)	F2 (36H)	F1 (35H)	CODE (34H)	CAP (33H)	GRAPH (32H)	CTRL (31H)	SHIFT (30H)	Row 6
CR (3FH)	SEL (3EH)	BS (3DH)	STOP (3CH)	TAB (3BH)	ESC (3AH)	F5 (39H)	F4 (38H)	Row 7
RIGHT (47H)	DOWN (46H)	UP (45H)	LEFT (44H)	DEL (43H)	INS (42H)	HOME (41H)	SPACE (40H)	Row 8
4 (4FH)	3 (4EH)	2 (4DH)	1 (4CH)	0 (4BH)	(4AH)	(49H)	(48H)	Row 9
. (57H)	, (56H)	- (55H)	9 (54H)	8 (53H)	7 (52H)	6 (51H)	5 (50H)	Row 10
7	6	5	4	3	2	1	0	Column

Figure 36: Key Numbers

The key number is then converted into a key code and placed in KEYBUF (1021H). When all eight possible bits have been processed the routine terminates.

```
Address... 0DA5H
```

This table contains the key codes of key numbers 00H to 2FH for various combinations of the control keys. A zero entry in the table means that no key code will be produced when that key is pressed:

```
        37H   36H   35H   34H   33H   32H   31H   30H    Row  0
        3BH   5DH   5BH   5CH   3DH   2DH   39H   38H    Row  1
NORMAL  62H   61H   9CH   2FH   2EH   2CH   60H   27H    Row  2
        6AH   69H   68H   67H   66H   65H   64H   63H    Row  3
        72H   71H   70H   6FH   6EH   6DH   6CH   6BH    Row  4
        7AH   79H   78H   77H   76H   75H   74H   73H    Row  5

        26H   5EH   25H   24H   23H   40H   21H   29H    Row  0
        3AH   7DH   7BH   7CH   2BH   5FH   28H   2AH    Row  1
```

```
SHIFT  42H  41H  9CH  3FH  3EH  3CH  7EH  22H   Row  2
       4AH  49H  48H  47H  46H  45H  44H  43H   Row  3
       52H  51H  50H  4FH  4EH  4DH  4CH  4BH   Row  4
       5AH  59H  58H  57H  56H  55H  54H  53H   Row  5

       FBH  F4H  BDH  EFH  BAH  ABH  ACH  09H   Row  0
       06H  0DH  01H  1EH  F1H  17H  07H  ECH   Row  1
GRAPH  11H  C4H  9CH  1DH  F2H  F3H  BBH  05H   Row  2
       C6H  DCH  13H  15H  14H  CDH  C7H  BCH   Row  3
       18H  CCH  DBH  C2H  1BH  0BH  C8H  DDH   Row  4
       0FH  19H  1CH  CFH  1AH  C0H  12H  D2H   Row  5

       00H  F5H  00H  00H  FCH  FDH  00H  0AH   Row  0
       04H  0EH  02H  16H  F0H  1FH  08H  00H   Row  1
SHIFT  00H  FEH  9CH  F6H  AFH  AEH  F7H  03H   Row  2
GRAPH  CAH  DFH  D6H  10H  D4H  CEH  C1H  FAH   Row  3
       A9H  CBH  D7H  C3H  D3H  0CH  C9H  DEH   Row  4
       F8H  AAH  F9H  D0H  D5H  C5H  00H  D1H   Row  5

       E1H  E0H  98H  9BH  BFH  D9H  9FH  EBH   Row  0
       B7H  DAH  EDH  9CH  E9H  EEH  87H  E7H   Row  1
CODE   97H  84H  9CH  A7H  A6H  86H  E5H  B9H   Row  2
       91H  A1H  B1H  81H  94H  8CH  8BH  8DH   Row  3
       93H  83H  A3H  A2H  A4H  E6H  B5H  B3H   Row  4
       85H  A0H  8AH  88H  95H  82H  96H  89H   Row  5

       00H  00H  9DH  9CH  BEH  9EH  ADH  D8H   Row  0
       B6H  EAH  E8H  00H  00H  00H  80H  E2H   Row  1
SHIFT  00H  8EH  9CH  A8H  00H  8FH  E4H  B8H   Row  2
CODE   92H  00H  B0H  9AH  99H  00H  00H  00H   Row  3
       00H  00H  E3H  00H  A5H  00H  B4H  B2H   Row  4
       00H  00H  00H  00H  00H  90H  00H  00H   Row  5

        7    6    5    4    3    2    1    0    Column
```

```
Address... 0EC5H
```

Control transfers to this routine, from 0FC3H, to complete decoding of the five function keys. The relevant entry in FNKFLG is first checked to determine whether the key is associated with an "`ON KEY GOSUB`" statement. If so, and provided that CURLIN shows the BASIC Interpreter to be in program mode, the relevant entry in TRPTBL is updated (0EF1H) and the routine terminates. If the key is not tied to an "`ON KEY GOSUB`" statement, or if the Interpreter is in direct mode, the string of characters associated with the function key is returned instead. The key number is multiplied by sixteen, as each string is sixteen characters long, and added to the starting address of the function key strings in the Workspace Area. Sequential characters are then taken from the string and placed in KEYBUF (0F55H) until the zero byte terminator is reached.

```
Address... 0EF1H
```

This routine is used to update a device's entry in TRPTBL when it has produced a BASIC program interrupt. On entry register pair HL points to the device's status byte in the table. Bit 0 of the status byte is checked first, if the device is not "`ON`" then the routine terminates with no action. Bit 2, the event flag, is then checked. If this is already set then the routine terminates, otherwise it is set to indicate that an event has occurred. Bit 1, the "`STOP`" flag, is then checked. If the device is stopped then the routine terminates with no further action. Otherwise ONGSBF is incremented to signal to the Interpreter Runloop that the event should now be processed.

```
Address... 0F06H
```

This section of the key decoder processes the HOME key only. The state of the SHIFT key is determined via row 6 of NEWKEY and the key code for HOME (0BH) or CLS (0CH) placed in KEYBUF (0F55H) accordingly.

```
Address... 0F10H
```

This section of the keyboard decoder processes key numbers 30H to 57H apart from the CAP, F1 to F5, STOP and HOME keys. The key number is simply used to look up the key code in the table at 1033H and this is then placed in KEYBUF (0F55H).

```
Address... 0F1FH
```

This section of the keyboard decoder processes the DEAD key found on European MSX machines. On UK machines the key in row 2, column 5 always generates the pound key code (9CH) shown in the table at 0DA5H. On European machines this table will have the key code FFH in the same locations. This key code only serves as a flag to indicate that the next key pressed, if it is a vowel, should be modified to produce an accented graphics character.

The state of the SHIFT and CODE keys is determined via row 6 of NEWKEY and one of the following placed in KANAST: 01H=DEAD, 02H=DEAD+SHIFT, 03H=DEAD+CODE, 04H=DEAD+SHIFT+CODE.

```
Address... 0F36H
```

This section of the keyboard decoder processes the CAP key. The current state of CAPST is inverted and control drops into the CHGCAP standard routine.

```
Address... 0F3DH
Name..... CHGCAP
Entry..... A=ON/OFF Switch
```

```
Exit...... None
Modifies.. AF
```

Standard routine to turn the Caps Lock LED on or off as determined by the contents of register A: 00H=On, NZ=Off. The LED is modified using the bit set/reset facility of the PPI Mode Port. As CAPST is not changed this routine does not affect the characters produced by the keyboard.

```
Address... 0F46H
```

This section of the keyboard decoder processes the STOP key. The state of the CTRL key is determined via row 6 of NEWKEY and the key code for STOP (04H) or CTRL/STOP (03H) produced as appropriate. If the CTRL/STOP code is produced it is copied to INTFLG, for use by the ISCNTC standard routine, and then placed in KEYBUF (0F55H). If the STOP code is produced it is also copied to INTFLG but is not placed in KEYBUF, instead only a click is generated (0F64H). This means that an application program cannot read the STOP key code via the ROM BIOS standard routines.

```
Address... 0F55H
```

This section of the keyboard decoder places a key code in KEYBUF and generates an audible click. The correct address in the keyboard buffer is first taken from PUTPNT and the code placed there. The address is then incremented (105BH). If it has wrapped round and caught up with GETPNT then the routine terminates with no further action as the keyboard buffer is full. Otherwise PUTPNT is updated with the new address.

CLIKSW and CLIKFL are then both checked to determine whether a click is required. CLIKSW is a general enable/disable switch while CLIKFL is used to prevent multiple clicks when the function keys are pressed. Assuming a click is required the Key Click output is set via the PPI Mode Port and, after a delay of 50 µs, control drops into the CHGSND standard routine.

```
Address... 0F7AH
Name...... CHGSND
Entry..... A=ON/OFF Switch
Exit...... None
Modifies.. AF
```

Standard routine to set or reset the Key Click output via the PPI Mode Port: 00H=Reset, NZ=Set. This audio output is AC coupled so absolute polarities should not be taken too seriously.

```
Address... 0F83H
```

This section of the keyboard decoder processes key numbers 00H to 2FH. The state of the SHIFT, GRAPH and CODE keys is determined via row 6 of NEWKEY and combined with the key number to

form a look-up address into the table at 0DA5H. The key code is then taken from the table. If it is zero the routine terminates with no further action, if it is FFH control transfers to the DEAD key processor (0F1FH). If the code is in the range 40H to 5FH or 60H to 7FH and the CTRL key is pressed then the corresponding control code is placed in KEYBUF (0F55H). If the code is in the range 01H to 1FH then a graphic header code (01H) is first placed in KEYBUF (0F55H) followed by the code with 40H added. If the code is in the range 61H to 7BH and CAPST indicates that caps lock is on then it is converted to upper case by subtracting 20H. Assuming that KANAST contains zero, as it always will on UK machines, then the key code is placed in KEYBUF (0F55H) and the routine terminates. On European MSX machines, with a DEAD key instead of a pound key, then the key codes corresponding to the vowels a, e, i, o, u may be further modified into graphics codes.

```
Address... 0FC3H
```

This section of the keyboard decoder processes the five function keys. The state of the SHIFT key is examined via row 6 of NEWKEY and five added to the key number if it is pressed. Control then transfers to 0EC5H to complete processing.

```
Address... 1021H
```

This routine searches the table at 1B97H to determine which group of keys the key number supplied in register C belongs to. The associated address is then taken from the table and control transferred to that section of the keyboard decoder. Note that the table itself is actually patched into the middle of the OUTDO standard routine as a result of the modifications made to the Japanese ROM.

```
Address... 1033H
```

This table contains the key codes of key numbers 30H to 57H other than the special keys CAP, F1 to F5, STOP and HOME. A zero entry in the table means that no key code will be produced when that key is pressed:

```
00H 00H 00H 00H 00H 00H 00H 00H Row 6
0DH 18H 08H 00H 09H 1BH 00H 00H Row 7
1CH 1FH 1EH 1DH 7FH 12H 0CH 20H Row 8
34H 33H 32H 31H 30H 00H 00H 00H Row 9
2EH 2CH 2DH 39H 38H 37H 36H 35H Row 10

 7   6   5   4   3   2   1   0  Column
```

```
Address... 105BH
```

This routine simply zeroes KANAST and then transfers control to 10C2H.

```
Address... 1061H
```

This table contains the graphics characters which replace the vowels a, e, i, o, u on European machines.

```
Address... 10C2H
```

This routine increments the keyboard buffer pointer, either PUTPNT or GETPNT, supplied in register pair HL. If the pointer then exceeds the end of the keyboard buffer it is wrapped back to the beginning.

```
Address... 10CBH
Name...... CHGET
Entry..... None
Exit...... A=Character from keyboard
Modifies.. AF, EI
```

Standard routine to fetch a character from the keyboard buffer. The buffer is first checked to see if already contains a character (0D6AH). If not the cursor is turned on (09DAH), the buffer checked repeatedly until a character appears (0D6AH) and then the cursor turned off (0A27H). The character is taken from the buffer using GETPNT which is then incremented (10C2H).

```
Address... 10F9H
Name...... CKCNTC
Entry..... None
Exit...... None
Modifies.. AF, EI
```

Standard routine to check whether the CTRL-STOP or STOP keys have been pressed. It is used by the BASIC Interpreter inside processor-intensive statements, such as "`WAIT`" and "`CIRCLE`", to check for program termination. Register pair HL is first zeroed and then control transferred to the ISCNTC standard routine. When the Interpreter is running register pair HL normally contains the address of the current character in the BASIC program text. If ISCNTC is CTRL-STOP terminated this address will be placed in OLDTXT by the "`STOP`" statement handler (63E6H) for use by a later "`CONT`" statement. Zeroing register pair HL beforehand signals to the "`CONT`" handler that termination occurred inside a statement and it will issue a "`Can't CONTINUE`" error if continuation is attempted.

```
Address... 1102H
Name...... WRTPSG
Entry..... A=Register number, E=Data byte
Exit...... None
Modifies.. EI
```

Standard routine to write a data byte to any of the sixteen PSG registers. The register selection number is written to the PSG Address Port and the data byte written to the PSG Data Write Port.

```
Address... 110EH
Name..... RDPSG
Entry..... A=Register number
Exit..... A=Data byte read from PSG
Modifies.. A
```

Standard routine to read a data byte from any of the sixteen PSG registers. The register selection number is written to the PSG Address Port and the data byte read from the PSG Data Read Port.

```
Address... 1113H
Name..... BEEP
Entry..... None
Exit..... None
Modifies.. AF, BC, E, EI
```

Standard routine to produce a beep via the PSG. Channel A is set to produce a tone of 1316Hz then enabled with an amplitude of seven. After a delay of 40 ms control transfers to the GICINI standard routine to reinitialize the PSG.

```
Address... 113BH
```

This routine is used by the interrupt handler to service a music queue. As there are three of these, each feeding a PSG channel, the queue to be serviced is specified by supplying its number in register A: 0=VOICAQ, 1=VOICBQ and 2=VOICCQ.

Each string in a "`PLAY`" statement is translated into a series of data packets by the BASIC Interpreter. These are placed in the appropriate queue followed by an end of data byte (FFH). The task of dequeueing the packets, decoding them and setting the PSG is left to the interrupt handler. The Interpreter is thus free to proceed immediately to the next statement without having to wait for notes to finish.

The first two bytes of any packet specify its byte count and duration. The three most significant bits of the first byte specify the number of bytes following the header in the packet. The remainder of the header specifies the event duration in 20 ms units. This duration count determines how long it will be before the next packet is read from the queue.

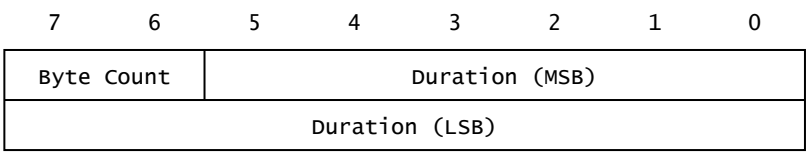


Figure 37: Packet Header

The packet header may be followed by zero or more blocks, in any order, containing frequency or amplitude information:

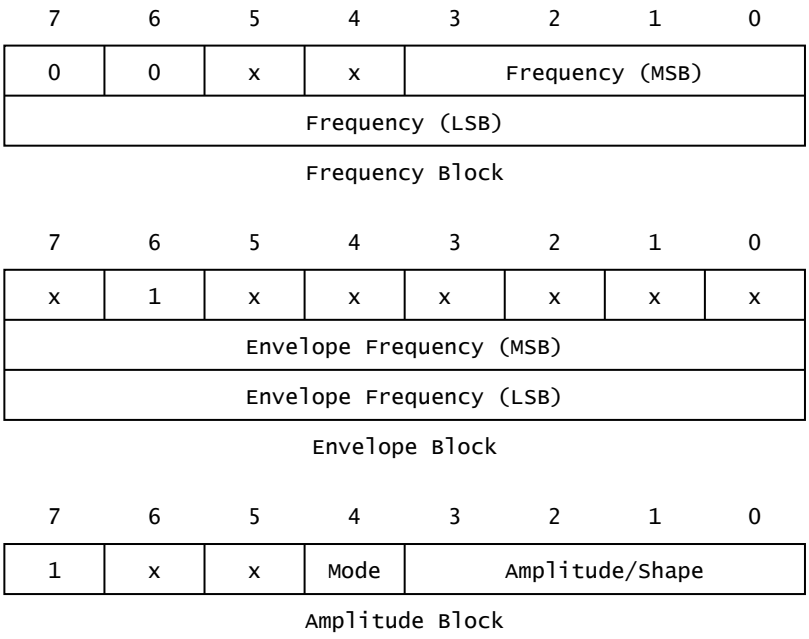


Figure 38: Packet Block Types

The routine first locates the current duration counter in the relevant voice buffer (VCBA, VCBB or VCBC) via the GETVCP standard routine and decrements it. If the counter has reached zero then the next packet must be read from the queue, otherwise the routine terminates.

The queue number is placed in QUEUEN and a byte read from the queue (11E2H). This is then checked to see if it is the end of data mark (FFH), if so the queue terminates (11B0H). Otherwise the byte count is placed in register C and the duration MSB in the relevant voice buffer. The second byte is read (11E2H) and the duration LSB placed in the relevant voice buffer. The byte count is then examined, if there are no bytes to follow the packet header the routine terminates. Otherwise successive bytes are read from the queue, and the appropriate action taken, until the byte count is exhausted.

If a frequency block is found then a second byte is read and both bytes written to PSG Registers 0 and 1, 2 and 3 or 4 and 5 depending on the queue number.

If an amplitude block is found the Amplitude and Mode bits are written to PSG Registers 8, 9 or 10 depending on the queue number. If the Mode bit is 1, selecting modulated rather than fixed amplitude, then the byte is also written to PSG Register 13 to set the envelope shape.

If an envelope block is found, or if bit 6 of an amplitude block is set, then a further two bytes are read from the queue and written to PSG Registers 11 and 12.

```
Address... 11B0H
```

This routine is used when an end of data mark (FFH) is found in one of the three music queues. An amplitude value of zero is written to PSG Register 8 9 or 10, depending on the queue number, to shut the channel down. The channel's bit in MUSICF is then reset and control drops into the STRTMS standard routine.

```
Address... 11C4H
Name...... STRTMS
Entry..... None
Exit...... None
Modifies.. AF, HL
```

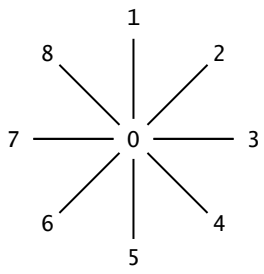
Standard routine used by the "`PLAY`" statement handler to initiate music dequeueing by the interrupt handler. MUSICF is first examined, if any channels are already running the routine terminates with no action. PLYCNT is then decremented, if there are no more "`PLAY`" strings queued up the routine terminates. Otherwise the three duration counters, in VCBA, VCBB and VCBC, are set to 0001H, so that the first packet of the new group will be dequeued at the next interrupt, and MUSICF is set to 07H to enable all three channels.

```
Address... 11E2H
```

This routine loads register A with the current queue number, from QUEUEN, and then reads a byte from that queue (14ADH).

```
Address... 11EEH
Name...... GTSTCK
Entry..... A=Joystick ID (0, 1 or 2)
Exit...... A=Joystick position code
Modifies.. AF, B, DE, HL, EI
```

Standard routine to read the position of a joystick or the four cursor keys. If the supplied ID is zero the state of the cursor keys is read via PPI Port B (1226H) and converted to a position code using the look-up table at 1243H. Otherwise joystick connector 1 or 2 is read (120CH) and the four direction bits converted to a position code using the look-up table at 1233H. The returned position codes are:




```
Address... 120CH
```

This routine reads the joystick connector specified by the contents of register A: 0=Connector 1, 1=Connector 2. The current contents of PSG Register 15 are read in then written back with the Joystick Select bit appropriately set. PSG Register 14 is then read into register A (110CH) and the routine terminates.

```
Address... 1226H
```

This routine reads row 8 of the keyboard matrix. The current contents of PPI Port C are read in then written back with the four Keyboard Row Select bits set for row 8. The column inputs are then read into register A from PPI Port B.

```
Address... 1253H
Name...... GTTRIG
Entry..... A=Trigger ID (0, 1, 2, 3 or 4)
Exit...... A=Status code
Modifies.. AF, BC, EI
```

Standard routine to check the joystick trigger or space key status. If the supplied ID is zero row 8 of the keyboard matrix is read (1226H) and converted to a status code. Otherwise joystick connector 1 or 2 is read (120CH) and converted to a status code. The selection IDs are:

```
0=SPACE KEY
1=JOY 1, TRIGGER A
2=JOY 2, TRIGGER A
3=JOY 1, TRIGGER B
4=JOY 2, TRIGGER B
```

The value returned is FFH if the relevant trigger is pressed and zero otherwise.

```
Address... 1273H
Name...... GTPDL
Entry..... A=Paddle ID (1 to 12)
Exit...... A=Paddle value (0 to 255)
Modifies.. AF, BC, DE, EI
```

Standard routine to read the value of any paddle attached to a joystick connector. Each of the six input lines (four direction plus two triggers) per connector can support a paddle so twelve are possible altogether. The paddles attached to joystick connector 1 have entry identifiers 1, 3, 5, 7, 9 and 11. Those attached to joystick connector 2 have entry identifiers 2, 4, 6, 8, 10 and 12. Each paddle is basically a one-shot pulse generator, the length of the pulse being controlled by a variable resistor. A start pulse is issued to the specified joystick connector via PSG Register 15. A count is then kept of how many times PSG Register 14 has to be read until the relevant input times out. Each unit increment represents an approximate period of 12 µs on an MSX machine with one wait state.

```
Address... 12ACH
Name..... GTPAD
Entry..... A=Function code (0 to 7)
Exit..... A=Status or value
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to access a touchpad attached to either of the joystick connectors. Available functions codes for joystick connector 1 are:

```
0=Return Activity Status
1=Return "X" coordinate
2=Return "Y" coordinate
3=Return Switch Status
```

Function codes 4 to 7 have the same effect with respect to joystick connector 2. The Activity Status function returns FFH if the Touchpad is being touched and zero otherwise. The Switch Status function returns FFH if the switch is being pressed and zero otherwise. The two coordinate request functions return the coordinates of the last location touched. These coordinates are actually stored in the Workspace Area variables PADX and PADY when a call with function code 0 or 4 detects activity. Note that these variables are shared by both joystick connectors.

```
Address... 1384H
Name..... STMOTR
Entry..... A=Motor ON/OFF code
Exit..... None
Modifies.. AF
```

Standard routine to turn the cassette motor relay on or off via PPI Port C: 00H=Off, 01H=On, FFH=Reverse current state.

```
Address... 1398H
Name..... NMI
Entry..... None
Exit..... None
Modifies.. None
```

Standard routine to process a Z80 Non Maskable Interrupt, simply returns on a standard MSX machine.

```
Address... 139DH
Name..... INIFNK
Entry..... None
```

```
Exit..... None
Modifies.. BC, DE, HL
```

Standard routine to initialize the ten function key strings to their power-up values. The one hundred and sixty bytes of data commencing at 13A9H are copied to the FNKSTR buffer in the Workspace Area.

```
Address... 13A9H
```

This area contains the power-up strings for the ten function keys. Each string is sixteen characters long, unused positions contain zeroes:

```
F1 to F5  F6 to F10
color     color 15,4,4 CR
auto      cload"
goto      cont CR
list      list. CR UP UP
run CR    run CLS CR
```

```
Address... 1449H
Name..... RDVDP
Entry..... None
Exit..... A=VDP Status Register contents
Modifies.. A
```

Standard routine to input the contents of the VDP Status Register by reading the Command Port. Note that reading the VDP Status Register will clear the associated flags and may affect the interrupt handler.

```
Address... 144CH
Name..... RSLREG
Entry..... None
Exit..... A=Primary Slot Register contents
Modifies.. A
```

Standard routine to input the contents of the Primary slot Register by reading PPI Port A.

```
Address... 144FH
Name..... WSLREG
Entry..... A=Value to write
Exit..... None
Modifies.. None
```

Standard routine to set the Primary Slot Register by writing to PPI Port A.

```
Address... 1452H
Name...... SNSMAT
Entry..... A=Keyboard row number
Exit...... A=Column data of keyboard row
Modifies.. AF, C, EI
```

Standard routine to read a complete row of the keyboard matrix. PPI Port C is read in then written back with the row number occupying the four Keyboard Row Select bits. PPI Port B is then read into register A to return the eight column inputs. The four miscellaneous control outputs of PPI Port C are unaffected by this routine.

```
Address... 145FH
Name...... ISFLIO
Entry..... None
Exit...... Flag NZ if file I/O active
Modifies.. AF
```

Standard routine to check whether the BASIC Interpreter is currently directing its input or output via an I/O buffer. This is determined by examining PTRFIL. It is normally zero but will contain a buffer FCB (File Control Block) address while statements such as " `PRINT#1` ", " `INPUT#1` ", etc. are being executed by the Interpreter.

```
Address... 146AH
Name...... DCOMPR
Entry..... HL, DE
Exit...... Flag NC if HL>DE, Flag Z if HL=DE, Flag C if HL<DE
Modifies.. AF
```

Standard routine used by the BASIC Interpreter to check the relative values of register pairs HL and DE.

```
Address... 1470H
Name...... GETVCP
Entry..... A=Voice number (0, 1, 2)
Exit...... HL=Address in voice buffer
Modifies.. AF, HL
```

Standard routine to return the address of byte 2 in the specified voice buffer (VCBA, VCBB or VCBC).

```
Address... 1474H
Name...... GETVC2
Entry..... L=Byte number (0 to 36)
```

```
Exit..... HL=Address in voice buffer
Modifies.. AF, HL
```

Standard routine to return the address of any byte in the voice buffer (VCBA, VCBB or VCBC) specified by the voice number in VOICEN.

```
Address... 148AH
Name..... PHYDIO
Entry..... None
Exit..... None
Modifies.. None
```

Standard routine for use by Disk BASIC, simply returns on standard MSX machines.

```
Address... 148EH
Name..... FORMAT
Entry..... None
Exit..... None
Modifies.. None
```

Standard routine for use by Disk BASIC, simply returns on standard MSX machines.

```
Address... 1492H
Name..... PUTQ
Entry..... A=Queue number, E=Data byte
Exit..... Flag Z if queue full
Modifies.. AF, BC, HL
```

Standard routine to place a data byte in one of the three music queues. The queue's get and put positions are first taken from QUETAB (14FAH). The put position is temporarily incremented and compared with the get position, if they are equal the routine terminates as the queue is full. Otherwise the queue's address is taken from QUETAB and the put position added to it. The data byte is placed at this location in the queue, the put position is incremented and the routine terminates. Note that the music queues are circular, if the get or put pointers reach the last position in the queue they wrap around back to the start.

```
Address... 14ADH
```

This routine is used by the interrupt handler to read a byte from one of the three music queues. The queue number is supplied in register A, the data byte is returned in register A and the routine returns Flag Z if the queue is empty. The queue's get and put positions are first taken from QUETAB (14FAH). If the putback flag is active then the data byte is taken from QUEBAK and the routine terminates (14D1H), this facility is unused in the current versions of the MSX ROM. The put position is then

compared with the get position, if they are equal the routine terminates as the queue is empty. Otherwise the queue's address is taken from QUETAB and the get position added to it. The data byte is read from this location in the queue, the get position is incremented and the routine terminates.

```
Address... 14DAH
```

This routine is used by the GICINI standard routine to initialize a queue's control block in QUETAB. The control block is first located in QUETAB (1504H) and the put, get and putback bytes zeroed. The size byte is set from register B and the queue address from register pair DE.

```
Address... 14EBH
Name...... LFTQ
Entry..... A=Queue number
Exit...... HL=Free space left in queue
Modifies.. AF, BC, HL
```

Standard routine to return the number of free bytes left in a music queue. The queue's get and put positions are taken from QUETAB (14FAH) and the free space determined by subtracting put from get.

```
Address... 14FAH
```

This routine returns a queue's control parameters from QUETAB, the queue number is supplied in register A. The control block is first located in QUETAB (1504H), the put position is then placed in register B, the get position in register C and the putback flag in register A.

```
Address... 1504H
```

This routine locates a queue's control block in QUETAB. The queue number is supplied in register A and the control block address returned in register pair HL. The queue number is simply multiplied by six, as there are six bytes per block, and added to the address of QUETAB as held in QUEUES.

```
Address... 1510H
Name...... GRPPRT
Entry..... A=Character
Exit...... None
Modifies.. EI
```

Standard routine to display a character on the screen in either Graphics Mode or Multicolour Mode, it is functionally equivalent to the CHPUT standard routine.

The CNVCHR standard routine is first used to check for a graphic character, if the character is a header code (01H) then the routine terminates with no action. If the character is a converted graphic one then the control code decoding section is skipped. Otherwise the character is checked to see if it is a control code. Only the CR code (0DH) is recognized (157EH), all other characters smaller than 20H are ignored.

Assuming the character is displayable its eight byte pixel pattern is copied from the ROM character set into the PATWRK buffer (0752H) and FORCLR copied to ATRBYT to set its colour. The current graphics coordinates are then taken from GRPACX and GRPACY and used to set the current pixel physical address via the SCALXY and MAPXYC standard routines.

The eight byte pattern in PATWRK is processed a byte at a time. At the start of each byte the current pixel physical address is obtained via the FETCHC standard routine and saved. The eight bits are then examined in turn. If the bit is a 1 the associated pixel is set by the SETC standard routine, if it is a 0 no action is taken. After each bit the current pixel physical address is moved right (16ACH). When the byte is finished, or the right hand edge of the screen is reached, the initial current pixel physical address is restored and moved down one position by the TDOWNC standard routine.

When the pattern is complete, or the bottom of the screen has been reached, GRPACX is updated. In Graphics Mode its value is increased by eight, in Multicolour Mode by thirty-two. If GRPACX then exceeds 255, the right hand edge of the screen, a CR operation is performed (157EH).

```
Address... 157EH
```

This routine performs the CR operation for the GRPPRT standard routine, this code functions as a combined CR,LF. GRPACX is zeroed and eight or thirty-two, depending on the screen mode, added to GRPACY. If GRPACY then exceeds 191, the bottom of the screen, it is set to zero.

GRPACX and GRPACY may be manipulated directly by an application program to compensate for the limited number of control functions available.

```
Address... 1599H
Name..... SCALXY
Entry..... BC=X coordinate, DE=Y coordinate
Exit..... Flag NC if clipped
Modifies.. AF
```

Standard routine to clip a pair of graphics coordinates if necessary. The BASIC Interpreter can produce coordinates in the range -32768 to +32767 even though this far exceeds the actual screen size. This routine modifies excessive coordinate values to fit within the physically realizable range. If the X coordinate is greater than 255 it is set to 255, if the Y coordinate is greater than 191 it is set to 191. If either coordinate is negative (greater than 7FFFH) it is set to zero. Finally if the screen is in Multicolour Mode both coordinates are divided by four as required by the MAPXYC standard routine.

```
Address... 15D9H
```

This routine is used to check the current screen mode, it returns Flag Z if the screen is in Graphics Mode.

```
Address... 15DFH
Name...... MAPXYC
Entry..... BC=X coordinate, DE=Y coordinate
Exit...... None
Modifies.. AF, D, HL
```

Standard routine to convert a graphics coordinate pair into the current pixel physical address. The location in the Character Pattern Table of the byte containing the pixel is placed in CLOC. The bit mask identifying the pixel within that byte is placed in CMASK. Slightly different conversion methods are used for Graphics Mode and Multicolour Mode, equivalent programs in BASIC are:

```
Graphics Mode

10 INPUT"X,Y Coordinates";X,Y
20 A=(Y\8)*256+(Y AND 7)+(X AND &HF8)
30 PRINT"ADDR=";HEX$(Base(12)+A);"H ";
40 RESTORE 100
50 FOR N=0 TO (X AND 7):READ M$: NEXT N
60 PRINT"MASK=";M$
70 GOTO 10
100 DATA 10000000
110 DATA 01000000
120 DATA 00100000
130 DATA 00010000
140 DATA 00001000
150 DATA 00000100
160 DATA 00000010
170 DATA 00000001
```

```
Multicolour Mode

10 INPUT"X,Y Coordinates";X,Y
20 X=X\4:Y-Y\4
30 A=(Y\8)*256+(Y AND 7)+(X*4 AND &HF8)
40 PRINT"ADDR=";HEX$(BASE(17)+A);"H ";
50 IF X MOD 2=0 THEN MS="11110000" ELSE MS="00001111"
60 PRINT"MASK=";M$
70 GOTO 10
```

The allowable input range for both programs is X=0 to 255 and Y=0 to 191. The data statements in the Graphics Mode program correspond to the eight byte mask table commencing at 160BH in the MSX ROM. Line 20 in the Multicolour Mode program actually corresponds to the division by four in the SCALXY standard routine. It is included to make the coordinate system consistent for both programs.

```
Address... 1639H
Name...... FETCHC
Entry..... None
Exit...... A=CMASK, HL=CLOC
Modifies.. A, HL
```

Standard routine to return the current pixel physical address, register pair HL is loaded from CLOC and register A from CMASK.

```
Address... 1640H
Name...... STOREC
Entry..... A=CMASK, HL=CLOC
Exit...... None
Modifies.. None
```

Standard routine to set the current pixel physical address, register pair HL is copied to CLOC and register A is copied to CMASK.

```
Address... 1647H
Name...... READC
Entry..... None
Exit...... A=Colour code of current pixel
Modifies.. AF, EI
```

Standard routine to return the colour of the current pixel. The VRAM physical address is first obtained via the FETCHC standard routine. If the screen is in Graphics Mode the byte pointed to by CLOC is read from the Character Pattern Table via the RDVRM standard routine. The required bit is then isolated by CMASK and used to select either the upper or lower four bits of the corresponding entry in the Colour Table.

If the screen is in Multicolour Mode the byte pointed to by CLOC is read from the Character Pattern Table via the RDVRM standard routine. CMASK is then used to select either the upper or lower four bits of this byte. The value returned in either case will be a normal VDP colour code from zero to fifteen.

```
Address... 1676H
Name...... SETATR
Entry..... A=Colour code
```

```
Exit..... Flag C if illegal code
Modifies.. Flags
```

Standard routine to set the graphics ink colour used by the SETC and NSETCX standard routines. The colour code, from zero to fifteen, is simply placed in ATRBYT.

```
Address... 167EH
Name..... SETC
Entry..... None
Exit..... None
Modifies.. AF, EI
```

Standard routine to set the current pixel to any colour, the colour code is taken from ATRBYT. The pixel's VRAM physical address is first obtained via the FETCHC standard routine. In Graphics Mode both the Character Pattern Table and Colour Table are then modified (186CH).

In Multicolour Mode the byte pointed to by CLOC is read from the Character Pattern Table by the RDVRM standard routine. The contents of ATRBYT are then placed in the upper or lower four bits, as determined by CMASK, and the byte written back via the WRTVRM standard routine

```
Address... 16ACH
```

This routine moves the current pixel physical address one position right. If the right hand edge of the screen is exceeded it returns with Flag C and the physical address is unchanged. In Graphics Mode CMASK is first shifted one bit right, if the pixel still remains within the byte the routine terminates. If CLOC is at the rightmost character cell (LSB=F8H to FFH) then the routine terminates with Flag C (175AH). Otherwise CMASK is set to 80H, the leftmost pixel, and 0008H added to CLOC.

In Multicolour Mode control transfers to a separate routine (1779H).

```
Address... 16C5H
Name..... RIGHTC
Entry..... None
Exit..... None
Modifies.. AF
```

Standard routine to move the current pixel physical address one position right. In Graphics Mode CMASK is first shifted one bit right, if the pixel still remains within the byte the routine terminates. Otherwise CMASK is set to 80H, the leftmost pixel, and 0008H added to CLOC. Note that incorrect addresses will be produced if the right hand edge of the screen is exceeded.

In Multicolour Mode control transfers to a separate routine (178BH).

```
Address... 16D8H
```

This routine moves the current pixel physical address one position left. If the left hand edge of the screen is exceeded it returns Flag C and the physical address is unchanged. In Graphics Mode CMASK is first shifted one bit left, if the pixel still remains within the byte the routine terminates. If CLOC is at the leftmost character cell (LSB=00H to 07H) then the routine terminates with Flag C (175AH). Otherwise CMASK is set to 01H, the rightmost pixel, and 0008H subtracted from CLOC.

In Multicolour Mode control transfers to a separate routine (179CH).

```
Address... 16EEH
Name..... LEFTC
Entry..... None
Exit..... None
Modifies.. AF
```

Standard routine to move the current pixel physical address one position left. In Graphics Mode CMASK is first shifted one bit left, if the pixel still remains within the byte the routine terminates. Otherwise CMASK is set to 01H, the leftmost pixel, and 0008H subtracted from CLOC. Note that incorrect addresses will be produced if the left hand edge of the screen is exceeded.

In Multicolour Mode control transfers to a separate routine (17ACH).

```
Address... 170AH
Name..... TDOWNC
Entry..... None
Exit..... Flag C if off screen
Modifies.. AF
```

Standard routine to move the current pixel physical address one position down. If the bottom edge of the screen is exceeded it returns Flag C and the physical address is unchanged. In Graphics Mode CLOC is first incremented, if it still remains within an eight byte boundary the routine terminates. If CLOC was in the bottom character row (CLOC>=1700H) then the routine terminates with Flag C (1759H). Otherwise 00F8H is added to CLOC.

In Multicolour Mode control transfers to a separate routine (17C6H).

```
Address... 172AH
Name..... DOWNC
Entry..... None
Exit..... None
Modifies.. AF
```

Standard routine to move the current pixel physical address one position down. In Graphics Mode CLOC is first incremented, if it still remains within an eight byte boundary the routine terminates. Otherwise 00F8H is added to CLOC. Note that incorrect addresses will be produced if the bottom edge of the screen is exceeded.

In Multicolour Mode control transfers to a separate routine (17DCH).

```
Address... 173CH
Name..... TUPC
Entry..... None
Exit..... Flag C if off screen
Modifies.. AF
```

Standard routine to move the current pixel physical address one position up. If the top edge of the screen is exceeded it returns with Flag C and the physical address is unchanged. In Graphics Mode CLOC is first decremented, if it still remains within an eight byte boundary the routine terminates. If CLOC was in the top character row (CLOC<0100H) then the routine terminates with Flag C. Otherwise 00F8H is subtracted from CLOC.

In Multicolour Mode control transfers to a separate routine (17E3H).

```
Address... 175DH
Name..... UPC
Entry..... None
Exit..... None
Modifies.. AF
```

Standard routine to move the current pixel physical address one position up. In Graphics Mode CLOC is first decremented, if it still remains within an eight byte boundary the routine terminates. Otherwise 00F8H is subtracted from CLOC. Note that incorrect addresses will be produced if the top edge of the screen is exceeded.

In Multicolour Mode control transfers to a separate routine (17F8H).

```
Address... 1779H
```

This is the Multicolour Mode version of the routine at 16ACH. It is identical to the Graphics Mode version except that CMASK is shifted four bit positions right and becomes F0H if a cell boundary is crossed.

```
Address... 178BH
```

This is the Multicolour Mode version of the RIGHTC standard routine. It is identical to the Graphics Mode version except that CMASK is shifted four bit positions right and becomes F0H if a cell

boundary is crossed.

```
Address... 179CH
```

This is the Multicolour Mode version of the routine at 16D8H. It is identical to the Graphics Mode version except that CMASK is shifted four bit positions left and becomes 0FH if a cell boundary is crossed.

```
Address... 17ACH
```

This is the Multicolour Mode version of the LEFTC standard routine. It is identical to the Graphics Mode version except that CMASK is shifted four bit positions left and becomes 0FH if a cell boundary is crossed.

```
Address... 17C6H
```

This is the Multicolour Mode version of the TDOWNC standard routine. It is identical to the Graphics Mode version except that the bottom boundary address is 0500H instead of 1700H. There is a bug in this routine which will cause it to behave unpredictably if MLTCGP, the Character Pattern Table base address, is changed from its normal value of zero. There should be an `EX DE,HL` instruction inserted at address 17CEH.

If the Character Pattern Table base is increased the routine will think it has reached the bottom of the screen when it actually has not. This routine is used by the "`PAINT`" statement so the following demonstrates the fault:

```
10 BASE(17)=&H1000
20 SCREEN 3
30 PSET(200,0)
40 DRAW"D180L100U180R100"
50 PAINT(150,90)
60 GOTO 60
```

```
Address... 17DCH
```

This is the Multicolour Mode version of the DOWNC standard routine, it is identical to the Graphics Mode version.

```
Address... 17E3H
```

This is the Multicolour Mode version of the TUPC standard routine. It is identical to the Graphics Mode version except that is has a bug as above, this time there should be an `EX DE,HL` instruction at address 17EBH.

If the Character Pattern Table base address is increased the routine will think it is within the table when it has actually exceeded the top edge of the screen. This may be demonstrated by removing the "`R100`" part of Line 40 in the previous program.

```
Address... 17F8H
```

This is the Multicolour Mode version of the UPC standard routine, it is identical to the Graphics Mode version.

```
Address... 1809H
Name...... NSETCX
Entry..... HL=Pixel fill count
Exit...... None
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to set the colour of multiple pixels horizontally rightwards from the current pixel physical address. Although its function can be duplicated by the SETC and RIGHTC standard routines this would result in significantly slower operation. The supplied pixel count should be chosen so that the right-hand edge of the screen is not passed as this will produce anomalous behaviour. The current pixel physical address is unchanged by this routine.

In Graphics Mode CMASK is first examined to determine the number of pixels to the right within the current character cell. Assuming the fill count is large enough these are then set (186CH). The remaining fill count is divided by eight to determine the number of whole character cells. Successive bytes in the Character Pattern Table are then zeroed and the corresponding bytes in the Colour Table set from ATRBYT to fill these whole cells. The remaining fill count is then converted to a bit mask, using the seven byte table at 185DH, and these pixels are set (186CH).

In Multicolour Mode control transfers to a separate routine (18BBH).

```
Address... 186CH
```

This routine sets up to eight pixels within a cell to a specified colour in Graphics Mode. ATRBYT contains the colour code, register pair HL the address of the relevant byte in the Character Pattern Table and register A a bit mask, 11100000 for example, where every 1 specifies a bit to be set.

If ATRBYT matches the existing 1 pixel colour in the corresponding Colour Table byte then each specified bit is set to 1 in the Character Pattern Table byte. If ATRBYT matches the existing 0 pixel colour in the corresponding Colour Table byte then each specified bit is set to 0 in the Character Pattern Table byte.

If ATRBYT does not match either of the existing colours in the Colour Table Byte then normally each specified bit is set to 1 in the Character Pattern Table byte and the 1 pixel colour changed in the Colour Table byte. However if this would result in all bits being set to 1 in the Character Pattern Table byte then each specified bit is set to 0 and the 0 pixel colour changed in the Colour Table byte.

```
Address... 18BBH
```

This is the Multicolour Mode version of the NSETCX standard routine. The SETC and RIGHTC standard routines are called until the fill count is exhausted. Speed of operation is not so important in Multicolour Mode because of the lower screen resolution and the consequent reduction in the number of operations required.

```
Address... 18C7H
Name..... GTASPC
Entry..... None
Exit..... DE=ASPCT1, HL=ASPCT2
Modifies.. DE, HL
```

Standard routine to return the "`CIRCLE`" statement default aspect ratios.

```
Address... 18CFH
Name..... PNTINI
Entry..... A=Boundary colour (0 to 15)
Exit..... Flag C if illegal colour
Modifies.. AF
```

Standard routine to set the boundary colour for the "`PAINT`" statement. In Multicolour Mode the supplied colour code is placed in BDRATR. In Graphics Mode BDRATR is copied from ATRBYT as it is not possible to have separate paint and boundary colours.

```
Address... 18E4H
Name..... SCANR
Entry..... B=Fill switch, DE=Skip count
Exit..... DE=Skip remainder, HL=Pixel count
Modifies.. AF, BC, DE, HL, EI
```

Standard routine used by the "`PAINT`" statement handler to search rightwards from the current pixel physical address until a colour code equal to BDRATR is found or the edge of the screen is reached. The terminating position becomes the current pixel physical address and the initial position is returned in CSAVEA and CSAVEM. The size of the traversed region is returned in register pair HL and FILNAM+1. The traversed region is normally filled in but this can be inhibited, in Graphics Mode only, by using an entry parameter of zero in register B. The skip count in register pair DE determines the maximum number of pixels of the required colour that may be ignored from the initial starting

position. This facility is used by the " `PAINT` " statement handler to search for gaps in a horizontal boundary blocking its upward progress.

```
Address... 197AH
Name...... SCANL
Entry..... None
Exit...... HL=Pixel count
Modifies.. AF, BC, DE, HL, EI
```

Standard routine to search leftwards from the current pixel physical address until a colour code equal to BDRATR is found or the edge of the screen is reached. The terminating position becomes the current pixel physical address and the size of the traversed region is returned in register pair HL. The traversed region is always filled in.

```
Address... 19C7H
```

This routine is used by the SCANL and SCANR standard routines to check the current pixel's colour against the boundary colour in BDRATR.

```
Address... 19DDH
Name...... TAPOOF
Entry..... None
Exit...... None
Modifies.. EI
```

Standard routine to stop the cassette motor after data has been written to the cassette. After a delay of 550 ms, on MSX machines with one wait state, control drops into the TAPIOF standard routine.

```
Address... 19E9H
Name...... TAPIOF
Entry..... None
Exit...... None
Modifies.. EI
```

Standard routine to stop the cassette motor after data has been read from the cassette. The motor relay is opened via the PPI Mode Port. Note that interrupts, which must be disabled during cassette data transfers for timing reasons, are enabled as this routine terminates.

```
Address... 19F1H
Name...... TAPOON
Entry..... A=Header length switch
Exit...... Flag C if CTRL-STOP termination
Modifies.. AF, BC, HL, DI
```

Standard routine to turn the cassette motor on, wait 550 ms for the tape to come up to speed and then write a header to the cassette. A header is a burst of HI cycles written in front of every data block so the baud rate can be determined when the data is read back.

The length of the header is determined by the contents of register A: 00H=Short header, NZ=Long header. The BASIC cassette statements " `SAVE` ", " `CSAVE` " and " `BSAVE` " all generate a long header at the start of the file, in front of the identification block, and thereafter use short headers between data blocks. The number of cycles in the header is also modified by the current baud rate so as to keep its duration constant:

```
1200 Baud SHORT ... 3840 Cycles ... 1.5 Seconds
1200 Baud LONG ... 15360 Cycles ... 6.1 Seconds
2400 Baud SHORT ... 7936 Cycles ... 1.6 Seconds
2400 Baud LONG ... 31744 Cycles ... 6.3 Seconds
```

After the motor has been turned on and the delay has expired the contents of HEADER are multiplied by two hundred and fifty-six and, if register A is non-zero, by a further factor of four to produce the cycle count. HI cycles are then generated (1A4DH) until the count is exhausted whereupon control transfers to the BREAKX standard routine. Because the CTRL-STOP key is only examined at termination it is impossible to break out part way through this routine.

```
Address... 1A19H
Name..... TAPOUT
Entry..... A=Data byte
Exit..... Flag C if CTRL-STOP termination
Modifies.. AF, B, HL
```

Standard routine to write a single byte of data to the cassette. The MSX ROM uses a software driven FSK (Frequency Shift Keyed) method for storing information on the cassette. At the 1200 baud rate this is identical to the Kansas City Standard used by the BBC for the distribution of BASICODE programs.

At 1200 baud each 0 bit is written as one complete 1200 Hz LO cycle and each 1 bit as two complete 2400 Hz HI cycles. The data rate is thus constant as 0 and 1 bits have the same duration. When the 2400 baud rate is selected the two frequencies change to 2400 Hz and 4800 Hz but the format is otherwise unchanged.

A byte of data is written with a 0 start bit (1A50H), eight data bits with the least significant bit first, and two 1 stop bits (1A40H). At the 1200 baud rate a single byte will have a nominal duration of 11 x 833 µs = 9.2 ms. After the stop bits have been written control transfers to the BREAKX standard routine to check the CTRL-STOP key. The byte 43H is shown below as it would be written to cassette:

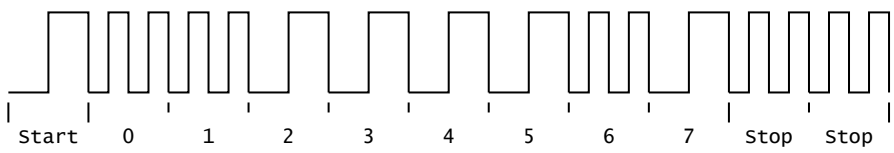


Figure 39: Cassette Data Byte

It is important not to leave too long an interval between bytes when writing data as this will increase the error rate. An inter-byte gap of 80 µs, for example, produces a read failure rate of approximately twelve percent. If a substantial amount of processing is required between each byte then buffering should be used to lump data into headered blocks. The BASIC "`SAVE`" format is of this type.

```
Address... 1A39H
```

This routine writes a single LO cycle with a length of approximately 816 µs to the cassette. The length of each half of the cycle is taken from LOW and control transfers to the general cycle generator (1A50H).

```
Address... 1A40H
```

This routine writes two HI cycles to the cassette. The first cycle is generated (1A4DH) followed by a 17 µs delay and then the second cycle (1A4DH).

```
Address... 1A4DH
```

This routine writes a single HI cycle with a length of approximately 396 µs to the cassette. The length of each half of the cycle is taken from HIGH and control drops into the general cycle generator.

```
Address... 1A50H
```

This routine writes a single cycle to the cassette. The length of the cycle's first half is supplied in register L and its second half in register H. The first length is counted down and then the Cas Out bit set via the PPI Mode Port. The second length is counted down and the Cas Out bit reset.

On all MSX machines the Z80 runs at a clock frequency of 3.579545 MHz (280 ns) with one wait state during the M1 cycle. As this routine counts every 16T states each unit increment in the length count represents a period of 4.47 µs. There is also a fixed overhead of 20.7 µs associated with the routine whatever the length count.

```
Address... 1A63H
Name..... TAPION
Entry..... None
```

```
Exit..... Flag C if CTRL-STOP termination
Modifies.. AF, BC, DE, HL, DI
```

Standard routine to turn the cassette motor on, read the cassette until a header is found and then determine the baud rate. Successive cycles are read from the cassette and the length of each one measured (1B34H). When 1,111 cycles have been found with less than 35 µs variation in their lengths a header has been located.

The next 256 cycles are then read (1B34H) and averaged to determine the cassette HI cycle length. This figure is multiplied by 1.5 and placed in LOWLIM where it defines the minimum acceptable length of a 0 start bit. The HI cycle length is placed in WINWID and will be used to discriminate between LO and HI cycles.

```
Address... 1ABCH
Name..... TAPIN
Entry..... None
Exit..... A=Byte read, Flag C if CTRL-STOP or I/O error
Modifies.. AF, BC, DE, L
```

Standard routine to read a byte of data from the cassette. The cassette is first read continuously until a start bit is found. This is done by locating a negative transition, measuring the following cycle length (1B1FH) and comparing this to see if it is greater than LOWLIM.

Each of the eight data bits is then read by counting the number of transitions within a fixed period of time (1B03H). If zero or one transitions are found it is a 0 bit, if two or three are found it is a 1 bit. If more than three transitions are found the routine terminates with Flag C as this is presumed to be a hardware error of some sort. After the value of each bit has been determined a further one or two transitions are read (1B23H) to retain synchronization. With an odd transition count one more will be read, with an even transition count two more.

```
Address... 1B03H
```

This routine is used by the TAPIN standard routine to count the number of cassette transitions within a fixed period of time. The window duration is contained in WINWID and is approximately 1.5 times the length of a HI cycle as shown below:

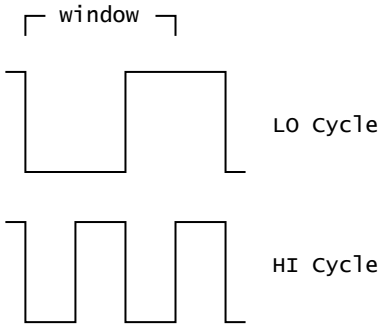


Figure 40: Cassette Window

The Cas Input bit is continuously sampled via PSG Register 14 and compared with the previous reading held in register E. Each time a change of state is found register C is incremented. The sampling rate is once every 17.3 µs so the value in WINWID, which was determined by the TAPION standard routine with a count rate of 11.45 µs, is effectively multiplied one and a half times.

```
Address... 1B1FH
```

This routine measures the time to the next cassette input transition. The Cassette Input bit is continuously sampled via PSG Register 14 until it changes from the state supplied in register E. The state flag is then inverted and the duration count returned in register C, each unit increment represents a period of 11.45 µs.

```
Address... 1B34H
```

This routine measures the length of a complete cassette cycle from negative transition to negative transition. The Cassette Input bit is sampled via PSG Register 14 until it goes to zero. The transition flag in register E is set to zero and the time to the positive transition measured (1B23H). The time to the negative transition is then measured (1B25H) and the total returned in register C.

```
Address... 1B45H
Name..... OUTDO
Entry..... A=Character to output
Exit..... None
Modifies.. EI
```

Standard routine used by the BASIC Interpreter to output a character to the current device. The ISFLIO standard routine is first used to check whether output is currently directed to an I/O buffer, if so control transfers to the sequential output driver (6C48H) via the CALBAS standard routine. If PRTFLG is zero control transfers to the CHPUT standard routine to output the character to the screen. Assuming the printer is active RAWPRT is checked. If this is non-zero the character is passed directly to the printer (1BABH), otherwise control drops into the OUTDLP standard routine.

```
Address... 1B63H
Name..... OUTDLP
Entry..... A=Character to output
Exit..... None
Modifies.. EI
```

Standard routine to output a character to the printer. If the character is a TAB code (09H) spaces are issued to the OUTDLP standard routine until LPTPOS is a multiple of eight (0, 8, 16 etc.). If the

character is a CR code (0DH) LPTPOS is zeroed if it is any other control code LPTPOS is unaffected, if it is a displayable character LPTPOS is incremented.

If NTMSXP is zero, meaning an MSX-specific printer is connected, the character is passed directly to the printer (1BABH). Assuming a normal printer is connected the CNVCHR standard routine is used to check for graphic characters. If the character is a header code (01H) the routine terminates with no action. If it is a converted graphic character it is replaced by a space, all other characters are passed to the printer (1BACH).

```
Address... 1B97H
```

This twenty byte table is used by the keyboard decoder to find the correct routine for a given key number:

KEY NUMBER	TO	FUNCTION
00H to 2FH	0F83H	Rows 0 to 5
30H to 32H	0F10H	SHIFT, CTRL, GRAPH
33H	0F36H	CAP
34H	0F10H	CODE
35H to 39H	0FC3H	F1 to F5
3AH to 3BH	0F10H	ESC, TAB
3CH	0F46H	STOP
3DH to 40H	0F10H	BS, CR, SEL, SPACE
41H	0F06H	HOME
42H to 57H	0F10H	INS, DEL, CURSOR

```
Address... 1BABH
```

This routine is used by the OUTDLP standard routine to pass a character to the printer. It is sent via the LPTOUT standard routine, if this returns Flag C control transfers to the "`Device I/O error`" generator (73B2H) via the CALBAS standard routine.

```
Address... 1BBFH
```

The following 2 KB contains the power-up character set. The first eight bytes contain the pattern for character code 00H, the second eight bytes the pattern for character code 01H and so on to character

code FFH.

```
Address... 23BFH
Name...... PINLIN
Entry..... None
Exit...... HL=Start of text, Flag C if CTRL-STOP termination
Modifies.. AF, BC, DE, HL, EI
```

Standard routine used by the BASIC Interpreter Mainloop to collect a logical line of text from the console. Control transfers to the INLIN standard routine just after the point where the previous line has been cut (23E0H).

```
Address... 23CCH
Name...... QINLIN
Entry..... None
Exit...... HL=Start of text, Flag C if CTRL-STOP termination
Modifies.. AF, BC, DE, HL, EI
```

Standard routine used by the "`INPUT`" statement handler to collect a logical line of text from the console. The characters "`?`" are displayed via the OUTDO standard routine and control drops into the INLIN standard routine.

```
Address... 23D5H
Name...... INLIN
Entry..... None
Exit...... HL=Start of text, Flag C if CTRL-STOP termination
Modifies.. AF, BC, DE, HL, EI
```

Standard routine used by the "`LINE INPUT`" statement handler to collect a logical line of text from the console. Characters are read from the keyboard until either the CR or CTRL-STOP keys are pressed. The logical line is then read from the screen character by character and placed in the Workspace Area text buffer BUF.

The current screen coordinates are first taken from CSRX and CSRY and placed in FSTPOS. The screen row immediately above the current one then has its entry in LINTTB made non-zero (0C29H) to stop it extending logically into the current row.

Each keyboard character read via the CHGET standard routine is checked (0919H) against the editing key table at 2439H. Control then transfers to one of the editing routines or to the default key handler (23FFH) as appropriate. This process continues until Flag C is returned by the CTRL-STOP or CR routines. Register pair HL is then set to point to the start of BUF and the routine terminates. Note that the carry, flag is cleared when Flag NZ is also returned to distinguish between a CR or protected CTRL-STOP termination and a normal CTRL-STOP termination.

```
Address... 23FFH
```

This routine processes all characters for the INLIN standard routine except the special editing keys. If the character is a TAB code (09H) spaces are issued (23FFH) until CSRX is a multiple of eight plus one (columns 1, 9, 17, 25, 33). If the character is a graphic header code (01H) it is simply echoed to the OUTDO standard routine. All other control codes smaller than 20H are echoed to the OUTDO standard routine after which INSFLG and CSTYLE are zeroed. For the displayable characters INSFLG is first checked and a space inserted (24F2H) if applicable before the character is echoed to the OUTDO standard routine.

```
Address... 2439H
```

This table contains the special editing keys recognized by the INLIN standard routine together with the relevant addresses:

CODE	TO	FUNCTION
08H	2561H	BS, backspace
12H	24E5H	INS, toggle insert mode
1BH	23FEH	ESC, no action
02H	260EH	CTRL-B, previous word
06H	25F8H	CTRL-F, next word
0EH	25D7H	CTRL-N, end of logical line
05H	25B9H	CTRL-E, clear to end of line
03H	24C5H	CTRL-STOP, terminate
0DH	245AH	CR, terminate
15H	25AEH	CTRL-U, clear line
7FH	2550H	DEL, delete character

```
Address... 245AH
```

This routine performs the CR operation for the INLIN standard routine. The starting coordinates of the logical line are found (266CH) and the cursor removed from the screen (0A2EH). Up to 254 characters are then read from the VDP VRAM (0BD8H) and placed in BUF. Any null codes (00H) are ignored, any characters smaller than 20H are replaced by a graphic header code (01H) and the character itself with 40H added. As the end of each physical row is reached LINTTB is checked

(0C1DH) to see whether the logical line extends to the next physical row. Trailing spaces are then stripped from BUF and a zero byte added as an end of text marker. The cursor is restored to the screen (09E1H) and its coordinates set to the last physical row of the logical line via the POSIT standard routine. A LF code is issued to the OUTDO standard routine, INSFLG is zeroed and the routine terminates with a CR code (0DH) in register A and Flag NZ,C. This CR code will be echoed to the screen by the INLIN standard routine mainloop just before it terminates.

```
Address... 24C5H
```

This routine performs the CTRL-STOP operation for the INLIN standard routine. The last physical row of the logical line is found by examining LINTTB (0C1DH), CSTYLE is zeroed, a zero byte is placed at the start of BUF and all music variables are cleared via the GICINI standard routine. TRPTBL is then examined (0454H) to see if an "`ON STOP`" statement is active, if so the cursor is reset (24AFH) and the routine terminates with Flag NZ,C. BASROM is then checked to see whether a protected ROM is running, if so the cursor is reset (24AFH) and the routine terminates with Flag NZ,C. Otherwise the cursor is reset (24B2H) and the routine terminates with Flag Z,C.

```
Address... 24E5H
```

This routine performs the INS operation for the INLIN standard routine. The current state of INSFLG is inverted and control terminates via the CSTYLE setting routine (242CH).

```
Address... 24F2H
```

This routine inserts a space character for the default key section of the INLIN standard routine. The cursor is removed (0A2EH) and the current cursor coordinates taken from CSRX and CSRY. The character at this position is read from the VDP VRAM (0BD8H) and replaced with a space (0BE6H). Successive characters are then copied one column position to the right until the end of the physical row is reached.

At this point LINTTB is examined (0C1DH) to determine whether the logical line is extended, if so the process continues on the next physical row. Otherwise the character taken from the last column position is examined, if this is a space the routine terminates by replacing the cursor (09E1H). Otherwise the physical row's entry in LINTTB is zeroed to indicate an extended logical line. The number of the next physical row is compared with the number of rows on the screen (0C32H). If the next row is the last one the screen is scrolled up (0A88H), otherwise a blank row is inserted (0AB7H) and the copying process continues.

```
Address... 2550H
```

This routine performs the DEL operation for the INLIN standard routine. If the current cursor position is at the rightmost column and the logical line is not extended no action is taken other than to write a space to the VDP VRAM (2595H). Otherwise a RIGHT code (1CH) is issued to the OUTDO standard routine and control drops into the BS routine.

```
Address... 2561H
```

This routine performs the BS operation for the INLIN standard routine. The cursor is first removed (0A2EH) and the cursor column coordinate decremented unless it is at the leftmost position and the previous row is not extended. Characters are then read from the VDP VRAM (0BD8H) and written back one position to the left (0BE6H) until the end of the logical line is reached. At this point a space is written to the VDP VRAM (0BE6H) and the cursor character is restored (09E1H).

```
Address... 25AEH
```

This routine performs the CTRL-U operation for the INLIN standard routine. The cursor is removed (0A2EH) and the start of the logical line located (266CH) and placed in CSRX and CSRY. The entire logical line is then cleared (25BEH).

```
Address... 25B9H
```

This routine performs the CTRL-E operation for the INLIN standard routine. The cursor is removed (0A2EH) and the remainder of the physical row cleared (0AEEH). This process is repeated for successive physical rows until the end of the logical line is found in LINTBB (0C1DH). The cursor is then restored (09E1H), INSFLG zeroed and CSTLYE reset to a block cursor (242DH).

```
Address... 25D7H
```

This routine performs the CTRL-N operation for the INLIN standard routine. The cursor is removed (0A2EH) and the last physical row of the logical line found by examination of LINTTB (0C1DH). Starting at the rightmost column of this physical row characters are read from the VDP VRAM (0BD8H) until a non-space character is found. The cursor coordinates are then set one column to the right of this position (0A5BH) and the routine terminates by restoring the cursor (25CDH).

```
Address... 25F8H
```

This routine performs the CTRL-F operation for the INLIN standard routine. The cursor is removed (0A2EH) and moved successively right (2624H) until a non-alphanumeric character is found. The cursor is then moved successively right (2624H) until an alphanumeric character is found. The routine terminates by restoring the cursor (25CDH).

```
Address... 260EH
```

This routine performs the CTRL-B operation for the INLIN standard routine. The cursor is removed (0A2EH) and moved successively left (2634H) until an alphanumeric character is found. The cursor is then moved successively left (2634H) until a non-alphanumeric character is found and then moved one position right (0A5BH). The routine terminates by restoring the cursor (25CDH).

```
Address... 2624H
```

This routine moves the cursor one position right (0A5BH), loads register D with the rightmost column number, register E with the bottom row number and then tests for an alphanumeric character at the cursor position (263DH).

```
Address... 2634H
```

This routine moves the cursor one position left (0A4CH), loads register D with the leftmost column number and register E with the top row number. The current cursor coordinates are compared with these values and the routine terminates Flag Z if the cursor is at this position. Otherwise the character at this position is read from the VDP VRAM (0BD8H) and checked to see if it is alphanumeric. If so the routine terminates Flag NZ,C otherwise it terminates Flag NZ,NC.

The alphanumeric characters are the digits "0" to "9" and the letters "A" to "Z" and "a" to "z". Also included are the graphics characters 86H to 9FH and A6H to FFH, these were originally Japanese letters and should have been excluded during the conversion to the UK ROM.

```
Address... 266CH
```

This routine finds the start of a logical line and returns its screen coordinates in register pair HL. Each physical row above the current one is checked via the LINTTB table (0C1DH) until a non-extended row is found. The row immediately below this on the screen is the start of the logical line and its row number is placed in register L. This is then compared with FSTPOS, which contains the row number when the INLIN standard routine was first entered, to see if the cursor is still on the same line. If so the column coordinate in register H is set to its initial position from FSTPOS. Otherwise register H is set to the leftmost position to return the whole line.

```
Address...2680H, JP to power-up initialize routine (7C76H).
Address...2683H, JP to the SYNCHR standard routine (558CH).
Address...2686H, JP to the CHRGTR standard routine (4666H).
Address...2689H, JP to the GETYPR standard routine (5597H).
```

5. ROM BASIC Interpreter

Microsoft BASIC has evolved over the years to its present position as the industry standard. It was originally written for the 8080 Microprocessor and even the MSX version is held in 8080 Assembly Language form. This process of continuous development means that there are less Z80-specific instructions than would be expected in a more modern program. It also means that numerous changes have been made and the result is a rather convoluted program. The structure of the Interpreter makes it unlikely that an application program will be able to use its many facilities. However most programs will need to cooperate with it to some extent so this chapter gives a detailed description of its operation.

There are four readily identifiable areas of importance within the Interpreter, the one most familiar to any user is the Mainloop (4134H). This collects numbered lines of text from the console and places them in order in the Program Text Area of memory until a direct statement is received.

The Runloop (4601H) is responsible for the execution of a program. It examines the first token of each program line and calls the appropriate routine to process the remainder of the statement. This continues until no more program text remains, control then returns to the Mainloop.

The analysis of numeric or string operands within a statement is performed by the Expression Evaluator (4C64H). Each expression is composed of factors, in turn analyzed by the Factor Evaluator (4DC7H), which are linked together by dyadic infix operators. As there are several types of operand, notably line numbers, which cannot form part of an expression in Microsoft BASIC the term "evaluated" is only used to refer to those that can. Otherwise a term such as "computed" will be used.

One point to note when examining the Interpreter in detail is that it contains a lot of trick code. The writers seem particularly fond of jumping into the middle of instructions to provide multiple entry points to a routine. As an example take the instruction:

```
3E D1       Normal: LD   A,0D1H
```

When encountered in the usual way this will of course load the accumulator with the value D1H. However if it is entered at "Normal" then it will be executed as a `POP DE` instruction. The Interpreter has many similarly obscure sections.

```
Address... 268CH
```

This routine is used by the Expression Evaluator to subtract two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. The second

operand's mantissa sign is inverted and control drops into the addition routine.

```
Address... 269AH
```

This routine is used by the Expression Evaluator to add two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. If the second operand is zero the routine terminates with no action, if the first operand is zero the second operand is copied to DAC (2F05H) and the routine terminates. The two exponents are compared, if they differ by more than 10^15 the routine terminates with the larger operand as the result. Otherwise the difference between the two exponents is used to align the mantissae by shifting the smaller one rightwards (27A3H), for example:

```
19.2100 = .1921*10^2 = .192100
+ .7436 = .7436*10^0 = .007436
```

If the two mantissa signs are equal the mantissae are then added (2759H), if they are different the mantissae are subtracted (276BH). The exponent of the result is simply the larger of the two original exponents. If an overflow was produced by addition the result mantissa is shifted right one digit (27DBH) and the exponent incremented. If leading zeroes were produced by subtraction the result mantissa is renormalized by shifting left (2797H). The guard byte is then examined and the result rounded up if the fifteenth digit is equal to or greater than five.

```
Address... 2759H
```

This routine adds the two double precision mantissae contained in DAC and ARG and returns the result in DAC. Addition commences at the least significant positions, DAC+7 and ARG+7, and proceeds two digits at a time for the seven bytes.

```
Address... 276BH
```

This routine subtracts the two double precision mantissae contained in DAC and ARG and returns the result in DAC. Subtraction commences at the guard bytes, DAC+8 and ARG+8, and proceeds two digits at a time for the eight bytes. If the result underflows it is corrected by subtracting it from zero and inverting the mantissa sign, for example:

```
0.17-0.85 = 0.32 = -0.68
```

```
Address... 2797H
```

This routine shifts the double precision mantissa contained in DAC one digit left.

```
Address... 27A3H
```

This routine shifts a double precision mantissa right. The number of digits to shift is supplied in register A, the address of the mantissa's most significant byte is supplied in register pair HL. The digit count is first divided by two to separate the byte and digit counts. The required number of whole bytes are then shifted right and the most significant bytes zeroed. If an odd number of digits was specified the mantissa is then shifted a further one digit right.

```
Address... 27E6H
```

This routine is used by the Expression Evaluator to multiply two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. If either operand is zero the routine terminates with a zero result (2E7DH). Otherwise the two exponents are added to produce the result exponent. If this is smaller than 10^-63 the routine terminates with a zero result, if it is greater than 10^63 an "`Overflow error`" is generated (4067H). The two mantissa signs are then processed to yield the sign of the result, if they are the same the result is positive, if they differ it is negative.

Even though the mantissae are in BCD format they are multiplied using the normal binary add and shift method. To accomplish this the first operand is successively multiplied by two (288AH) to produce the constants X*80, X*40, X*20, X*10, X*8, X*4, X*2, and X in the HOLD8 buffer. The second operand remains in ARG and DAC is zeroed to function as the product accumulator. Multiplication proceeds by taking successive pairs of digits from the second operand starting with the least significant pair. For each 1 bit in the digit pair the appropriate multiple of the first operand is added to the product. As an example the single multiplication 1823*96 would produce:

```
1823*10010110=(1823*80)+(1823*10)+(1823*4)+(1823*2)
```

As each digit pair is completed the product is shifted two digits right. When all seven digit pairs have been processed the routine terminates by renormalizing and rounding up the product (26FAH).

The time required for a multiplication depends largely upon the number of 1 bits in the second operand. The worst case, when all the digits are sevens, can take up to 11 ms compared to the average of approximately 7 ms.

```
Address... 288AH
```

This routine doubles a double precision mantissa three successive times to produce the products X*2, X*4 and X*8. The address of the mantissa's least significant byte is supplied in register pair DE. The products are stored at successively lower addresses commencing immediately below the operand.

```
Address... 289FH
```

This routine is used by the Expression Evaluator to divide two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. If the first operand is zero the routine terminates with a zero result if the second operand is zero a "`Division by zero`" error is generated (4058H). Otherwise the two exponents are subtracted to produce the result exponent and the two mantissa signs processed to yield the sign of the result. If they are the same the result is positive, if they differ it is negative.

The mantissae are divided using the normal long division method. The second operand is repeatedly subtracted from the first until underflow to produce a single digit of the result. The second operand is then added back to restore the remainder (2761H), the digit is stored in HOLD and the first operand is shifted one digit left. When the first operand has been completely shifted out the result is copied from HOLD to DAC then renormalized and rounded up (2883H). The time required for a division reaches a maximum of approximately 25 ms when the first operand is composed largely of nines and the second operand of ones. This will require the greatest number of subtractions.

```
Address... 2993H
```

This routine is used by the Factor Evaluator to apply the "`COS`" function to a double precision operand contained in DAC. The operand is first multiplied (2C3BH) by 1/(2*PI) so that unity corresponds to a complete 360 degree cycle. The operand then has 0.25 (90 degrees) subtracted (2C32H), its mantissa sign is inverted (2E8DH) and control drops into the "`SIN`" routine.

```
Address... 29ACH
```

This routine is used by the Factor Evaluator to apply the "`SIN`" function to a double precision operand contained in DAC. The operand is first multiplied (2C3BH) by 1/(2*PI) so that unity corresponds to a complete 360 degree cycle. As the function is periodic only the fractional part of the operand is now required. This is extracted by pushing the operand (2CCCH) obtaining the integer part (30CFH) and copying it to ARG (2C4DH), popping the whole operand to DAC (2CE1H) and then subtracting the integer part (268CH).

The first digit of the mantissa is then examined to determine the operand's quadrant. If it is in the first quadrant it is unchanged. If it is in the second quadrant it is subtracted from 0.5 (180 degrees) to reflect it about the Y axis. If it is in the third quadrant it is subtracted from 0.5 (180 degrees) to reflect it about the X axis. If it is in the fourth quadrant 1.0 (360 degrees) is subtracted to reflect it about both axes. The function is then computed by polynomial approximation (2C88H) using the list of coefficients at 2DEFH. These are the first eight terms in the Taylor series X-(X^3/3!)+(X^5/5!)-(X^7/7!) ... with the coefficients multiplied by successive factors of 2*PI to compensate for the initial scaling.

```
Address... 29FBH
```

This routine is used by the Factor Evaluator to apply the " `TAN` " function to a double precision operand contained in DAC. The function is computed using the trigonometric identity TAN(X) = SIN(X)/COS(X).

```
Address... 2A14H
```

This routine is used by the Factor Evaluator to apply the " `ATN` " function to a double precision operand contained in DAC. The function is computed by polynomial approximation (2C88H) using the list of coefficients at 2E30H. These are the first eight terms in the Taylor series X-(X^3/3)+(X^5/5)-(X^7/7) ... with the coefficients modified slightly to telescope the series.

```
Address... 2A72H
```

This routine is used by the Factor Evaluator to apply the " `LOG` " function to a double precision operand contained in DAC. The function is computed by polynomial approximation using the list of coefficients at 2DA5H.

```
Address... 2AFFH
```

This routine is used by the Factor Evaluator to apply the " `SQR` " function to a double precision operand contained in DAC. The function is computed using the Newton-Raphson process, an equivalent BASIC program is:

```
10 INPUT"NUMBER";X
20 GUESS=10
30 FOR N=1 To 7
40 GUESS=(GUESS+X/GUESS)/2
50 NEXT N
60 PRINT GUESS
70 PRINT SQR(X)
```

The above program uses a fixed initial guess. While this is accurate over a limited range maximum accuracy will only be attained if the initial guess is near the root. The method used by the ROM is to halve the exponent, with rounding up, and then to divide the first two digits of the operand by four and increment the first digit.

```
Address... 2B4AH
```

This routine is used by the Factor Evaluator to apply the " `EXP` " function to a double precision operand contained in DAC. The operand is first multiplied by 0.4342944819, which is LOG(e) to Base 10, so that the problem becomes computing 10^X rather than e^X. This results in considerable simplification as the integer part can be dealt with easily. The function is then computed by polynomial approximation using the list of coefficients at 2D6BH.

```
Address... 2BDFH
```

This routine is used by the Factor Evaluator to apply the " `RND` " function to a double precision operand contained in DAC. If the operand is zero the current random number is copied to DAC from RNDX and the routine terminates. If the operand is negative it is copied to RNDX to set the current random number. The new random number is produced by copying RNDX to HOLD, the constant at 2CF9H to ARG, the constant at 2CF1H to DAC and then multiplying (282EH). The fourteen least significant digits of the double length product are copied to RNDX to form the mantissa of the new random number. The exponent byte in DAC is set to 10^0 to return a value in the range 0 to 1.

```
Address... 2C24H
```

This routine is used by the " `NEW` ", " `CLEAR` " and " `RUN` " statement handlers to initialize RNDX with the constant at 2D01H.

```
Address... 2C2CH
```

This routine adds the constant whose address is supplied in register pair HL to the double precision operand contained in DAC.

```
Address... 2C32H
```

This routine subtracts the constant whose address is supplied in register pair HL from the double precision operand contained in DAC.

```
Address... 2C3BH
```

This routine multiplies the double precision operand contained in DAC by the constant whose address is supplied in register pair HL.

```
Address... 2C41H
```

This routine divides the double precision operand contained in DAC by the constant whose address is supplied in register pair HL.

```
Address... 2C47H
```

This routine performs the relation operation on the double precision operand contained in DAC and the constant whose address is supplied in register pair HL.

```
Address... 2C4DH
```

This routine copies an eight byte double precision operand from DAC to ARG.

```
Address... 2C59H
```

This routine copies an eight byte double precision operand from ARG to DAC.

```
Address... 2C6FH
```

This routine exchanges the eight bytes in DAC with the eight bytes currently on the bottom of the Z80 stack.

```
Address... 2C80H
```

This routine inverts the mantissa sign of the operand contained in DAC (2E8DH). The same address is then pushed onto the stack to restore the sign when the caller terminates.

```
Address... 2C88H
```

This routine generates an odd series based on the double precision operand contained in DAC. The series is of the form:

```
X^1*(Kn)+X^3*(Kn-1)+x^5*(Kn-2)+X^5*(Kn-3) ...
```

The address of the coefficient list is supplied in register pair HL. The first byte of the list contains the coefficient count, the double precision coefficients follow with K1 first and Kn last. The even series is generated (2C9AH) and multiplied (27E6H) by the original operand.

```
Address... 2C9AH
```

This routine generates an even series based on the double precision operand contained in DAC. The series is of the form:

```
X^0*(Kn)+x^2*(Kn-1)+x^4*(Kn-2)+x^6*(Kn-3) ...
```

The address of the coefficient list is supplied in register pair HL. The first byte of the list contains the coefficient count, the double precision coefficients follow with K1 first and Kn last. The method used to compute the polynomial is known as Horner's method. It only requires one multiplication and one addition per term, the BASIC equivalent is:

```
10 X=X*X
20 PRODUCT=0
30 RESTORE 100
40 READ COUNT
50 FOR N=1 TO COUNT
60 READ K
70 PRODUCT= ( PRODUCT*X ) +K
80 NEXT N
90 END
100 DATA 8
110 DATA Kn-7
120 DATA Kn-6
130 DATA Kn-5
140 DATA Kn-4
150 DATA Kn-3
160 DATA Kn-2
170 DATA Kn-1
180 DATA Kn
```

The polynomial is processed from the final coefficient through to the first coefficient so that the partial product can be used to save unnecessary operations.

```
Address... 2CC7H
```

This routine pushes an eight byte double precision operand from ARG onto the Z80 stack.

```
Address... 2CCCH
```

This routine pushes an eight byte double precision operand from DAC onto the Z80 stack.

```
Address... 2CDCH
```

This routine pops an eight byte double precision operand from the Z80 stack into ARG.

```
Address... 2CE1H
```

This routine pops an eight byte double precision operand from the Z80 stack into DAC.

```
Address... 2CF1H
```

This table contains the double precision constants used by the math routines. The first three constants have zero in the exponent position as they are in a special intermediate form used by the random number generator.

ADDRESS	CONSTANT		ADDRESS	CONSTANT	
2CF1H	.14389820420821	RND	2DAEH	6.2503651127908	
2CF9H	.21132486540519	RND	2DB6H	-13.682370241503	
2D01H	.40649651372358		2DBEH	8.5167319872389	
2D09H	.43429448190324	LOG(e)	2DC6H	5	LOG
2D11H	.50000000000000		2DC7H	1.0000000000000	
2D13H	.00000000000000		2DCFH	-13.210478350156	
2D1BH	1.0000000000000		2DD7H	47.925256043873	
2D23H	.25000000000000		2DDFH	-64.906682740943	
2D2BH	3.1622776601684	SQR(10)	2DE7H	29.415750172323	
2D33H	.86858896380650	2^LOG(e)	2DEFH	8	SIN
2D3BH	2.3025850929940	1/LOG(e)	2DF0H	-.69215692291809	
2D43H	1.5707963267949	PI/2	2DF8H	3.8172886385771	
2D4BH	.26794919243112	TAN(PI/12)	2E00H	-15.094499474801	
2D53H	1.7320508075689	TAN(PI/3)	2E08H	42.058689667355	
2D5BH	.52359877559830	PI/6	2E10H	-76.705859683291	
2D63H	.15915494309190	1/(2^PI)	2E18H	81.605249275513	
2D6BH	4	EXP	2E20H	-41.341702240398	
2D6CH	1.0000000000000		2E28H	6.2831853071796	
2D74H	159.37415236031		2E30H	8	ATN
2D7CH	2709.3169408516		2E31H	-.05208693904000	
2D84H	4497.6335574058		2E39H	.07530714913480	

ADDRESS	CONSTANT		ADDRESS	CONSTANT	
2D8CH	3	EXP	2E41H	-.09081343224705	
2D8DH	18.312360159275		2E49H	.11110794184029	
2D95H	831.40672129371		2E51H	-.14285708554884	
2D9DH	5178.0919915162		2E59H	.19999999948967	
2DA5H	4	LOG	2E61H	-.33333333333160	
2DA6H	-.71433382153226		2E69H	1.0000000000000	

```
Address... 2E71H
```

This routine returns the mantissa sign of a Floating Point operand contained in DAC. The exponent byte is tested and the result returned in register A and the flags:

```
Zero ....... A=00H, Flag Z,NC
Positive ... A=01H, Flag NZ,NC
Negative ... A=FFH, Flag NZ,C
```

```
Address... 2E7DH
```

This routine simply zeroes the exponent byte in DAC.

```
Address... 2E82H
```

This routine is used by the Factor Evaluator to apply the " `ABS` " function to an operand contained in DAC. The operand's sign is first checked (2EA1H), if it is positive the routine simply terminates. The operand's type is then checked via the GETYPR standard routine. If it is a string a " `Type mismatch` " error is generated (406DH). If it is an integer it is negated (322BH). If it is a double precision or single precision operand the mantissa sign bit in DAC is inverted.

```
Address... 2E97H
```

This routine is used by the Factor Evaluator to apply the " `SGN` " function to an operand contained in DAC. The operand's sign is checked (2EA1H), extended into register pair HL and then placed in DAC as an integer:

```
Zero ....... 0000H
Positive ... 0001H
```

```
Negative ... FFFFH
```

```
Address... 2EA1H
```

This routine returns the sign of an operand contained in DAC. The operands type is first checked via the GETYPR standard routine. If it is a string a "`Type mismatch`" error is generated (406DH). If it is a single precision or double precision operand the mantissa sign is examined (2E71H). If it is an integer its value is taken from DAC+2 and translated into the flags shown at 2E71H.

```
Address... 2EB1H
```

This routine pushes a four byte single precision operand from DAC onto the Z80 stack.

```
Address... 2EC1H
```

This routine copies the contents of registers C, B, E and D to DAC.

```
Address... 2ECCH
```

This routine copies the contents of DAC to registers C, B, E and D.

```
Address... 2ED6H
```

This routine loads registers C, B, E and D from upwardly sequential locations starting at the address supplied in register pair HL.

```
Address... 2EDFH
```

This routine loads registers E, D, C and B from upwardly sequential locations starting at the address supplied in register pair HL.

```
Address... 2EE8H
```

This routine copies a single precision operand from DAC to the address supplied in register pair HL.

```
Address... 2EEFH
```

This routine copies any operand from the address supplied in register pair HL to ARG. The length of the operand is contained in VALTYP: 2=Integer, 3=String, 4=Single Precision, 8=Double Precision.

```
Address... 2F05H
```

This routine copies any operand from ARG to DAC. The length of the operand is contained in VALTYP: 2=Integer, 3=String, 4=Single Precision, 8=Double Precision.

```
Address... 2F0DH
```

This routine copies any operand from DAC to ARG. The length of the operand is contained in VALTYP: 2=Integer, 3=String, 4=Single Precision, 8=Double Precision.

```
Address... 2F21H
```

This routine is used by the Expression Evaluator to find the relation (<>=) between two single precision operands. The first operand is contained in registers C, B, E and D and the second in DAC. The result is returned in register A and the flags:

```
Operand 1=Operand 2 ... A=00H, Flag Z,NC
Operand 1<Operand 2 ... A=01H, Flag NZ,NC
Operand 1>Operand 2 ... A=FFH, Flag NZ,C
```

It should be noted that for relational operators the Expression Evaluator regards maximally negative numbers as small and maximally positive numbers as large.

```
Address... 2F4DH
```

This routine is used by the Expression Evaluator to find the relation (<>=) between two integer operands. The first operand is contained in register pair DE and the second in register pair HL. The results are as for the single precision version (2F21H).

```
Address... 2F83H
```

This routine is used by the Expression Evaluator to find the relation (<>=) between two double precision operands. The first operand is contained in DAC and the second in ARG. The results are as for the single precision version (2F21H).

```
Address... 2F8AH
```

This routine is used by the Factor Evaluator to apply the "`CINT`" function to an operand contained in DAC. The operand type is first checked via the GETYPR standard routine, if it is already integer the routine simply terminates. If it is a string a "`Type mismatch`" error is generated (406DH). If it is a

single precision or double precision operand it is converted to a signed binary integer in register pair DE (305DH) and then placed in DAC as an integer. Out of range values result in an " `Overflow` " error (4067H).

```
Address... 2FA2H
```

This routine checks whether DAC contains the single precision operand -32768, if so it replaces it with the integer equivalent 8000H. This step is required during numeric input conversion (3299H) because of the asymmetric integer number range.

```
Address... 2FB2H
```

This routine is used by the Factor Evaluator to apply the " `CSNG` " function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is already single precision the routine simply terminates. If it is a string a " `Type mismatch` " error is generated (406DH). If it is double precision VALTYP is changed (3053H) and the mantissa rounded up from the seventh digit (2741H). If the operand is an integer it is converted from binary to a maximum of five BCD digits by successive divisions using the constants 10000, 1000, 100, 10, 1. These are placed in DAC to form the single precision mantissa. The exponent is equal to the number of significant digits in the mantissa. For example if there are five the exponent would be 10^5.

```
Address... 3030H
```

This table contains the five constants used by the " `CSNG` " routine: -10000, -1000, -100, -10, -1

```
Address... 303AH
```

This routine is used by the Factor Evaluator to apply the " `CDBL` " function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is already double precision the routine simply terminates. If it is a string a " `Type mismatch` " error is generated (406DH). If it is an integer it is first converted to single precision (2FC8H), the eight least significant digits are then zeroed and VALTYP set to 8.

```
Address... 3058H
```

This routine checks that the current operand is a string type, if not a " `Type mismatch` " error is generated (406DH).

```
Address... 305DH
```

This routine is used by the "`CINT`" routine (2F8AH) to convert a BCD single precision or double precision operand into a signed binary integer in register pair DE, it returns Flag C if an overflow has occurred. Successive digits are taken from the mantissa and added to the product starting with the most significant one. After each addition the product is multiplied by ten. The number of digits to process is determined by the exponent, for example five digits would be taken with an exponent of 10^5. Finally the mantissa sign is checked and the product negated (3221H) if necessary.

```
Address... 30BEH
```

This routine is used by the Factor Evaluator to apply the "`FIX`" function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is an integer the routine simply terminates. The mantissa sign is then checked (2E71H), if it is positive control transfers to the "`INT`" routine (30CFH). Otherwise the sign is inverted to positive, the "`INT`" function is performed (30CFH) and the sign restored to negative.

```
Address... 30CFH
```

This routine is used by the Factor Evaluator to apply the "`INT`" function to an operand contained in DAC. The operand's type is first checked via the GETYPR standard routine, if it is an integer the routine simply terminates. The number of fractional digits is determined by subtracting the exponent from the type's digit count, 6 for single precision, 14 for double precision.

If the mantissa sign is positive these fractional digits are simply zeroed. If the mantissa sign is negative each fractional digit is examined before it is zeroed. If all the digits were previously zero the routine simply terminates. Otherwise -1.0 is added to the operand by the single precision addition routine (324EH) or the double precision addition routine (269AH). It should be noted that an operand's type is not normally changed by the "`CINT`" function.

```
Address... 314AH
```

This routine multiplies the unsigned binary integers in register pairs BC and DE, the result is returned in register pair DE. The standard shift and add method is used, the product is successively multiplied by two and register pair BC added to it for every 1 bit in register pair DE. The routine is used by the Variable search routine (5EA4H) to compute an element's position within an Array, a "`Subscript out of range`" error is generated (601DH) if an overflow occurs.

```
Address... 3167H
```

This routine is used by the Expression Evaluator to subtract two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The second operand is negated (3221H) and control drops into the addition routine.

```
Address... 3172H
```

This routine is used by the Expression Evaluator to add two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The signed binary operands are normally just added and placed in DAC. However, if an overflow has occurred both operands are converted to single precision (2FCBH) and control transfers to the single precision adder (324EH). An overflow has occurred when both operands are of the same sign and the result is of the opposite sign, for example:

```
30000+15000=-20536
```

```
Address... 3193H
```

This routine is used by the Expression Evaluator to multiply two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The two operand signs are saved temporarily and both operands made positive (3215H). Multiplication proceeds using the standard binary shift and add method with register pair HL as the product accumulator, register pair BC containing the first operand and register pair DE the second. If the product exceeds 7FFFH at any time during multiplication both operands are converted to single precision (2FCBH) and control transfers to the single precision multiplier (325CH). Otherwise the initial signs are restored and, if they differ, the product negated before being placed in DAC as an integer (321DH).

```
Address... 31E6H
```

This routine is used by the Expression Evaluator to integer divide () two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. If the second operand is zero a "`Division by zero`" error is generated (4058H), otherwise the two operand signs are saved and both operands made positive (3215H). Division proceeds using the standard binary shift and subtract method with register pair HL containing the remainder, register pair BC the second operand and register pair DE the first operand and the product. When division is complete the initial signs are restored and, if they differ, the product is negated before being placed in DAC as an integer (321DH).

```
Address... 3215H
```

This routine is used to make two signed binary integers, in register pairs HL and DE, positive. Both the initial operand signs are returned as a flag in bit 7 of register B: 0=Same, 1=Different. Each operand is then examined and, if it is negative, made positive by subtracting it from zero.

```
Address... 322BH
```

This routine is used by the "`ABS`" function to make a negative integer contained in DAC positive. The operand is taken from DAC, negated and then placed back in DAC (3221H). If the operand's value is 8000H it is converted to single precision (2FCCH) as there is no integer of value +32768.

```
Address... 323AH
```

This routine is used by the Expression Evaluator to "`MOD`" two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The sign of the first operand is saved and the two operands divided (31E6H). As the remainder is returned doubled by the division process register pair DE is shifted one place right to restore it. The sign of the first operand is then restored and, if it is negative, the remainder is negated before being placed in DAC as an integer (321DH).

```
Address... 324EH
```

This routine is used by the Expression Evaluator to add two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first operand is copied to ARG (3280H), the second operand is converted to double precision (3042H) and control transfers to the double precision adder (269AH).

```
Address... 3257H
```

This routine is used by the Expression Evaluator to subtract two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The second operand is negated (2E8DH) and control transfers to the single precision adder (324EH).

```
Address... 325CH
```

This routine is used by the Expression Evaluator to multiply two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first operand is copied to ARG (3280H), the second operand is converted to double precision (3042H) and control transfers to the double precision multiplier (27E6H).

```
Address... 3265H
```

This routine is used by the Expression Evaluator to divide two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first and second operands are exchanged so that the first is in DAC and the second in the registers.

The second operand is then copied to ARG (3280H), the first operand is converted to double precision (3042H) and control transfers to the double precision divider (289FH).

```
Address... 3280H
```

This routine copies the single precision operand contained in registers C, B, E and D to ARG and then zeroes the four least significant bytes.

```
Address... 3299H
```

This routine converts a number in textual form to one of the standard internal numeric types, it is used during tokenization and by the "`VAL`", "`INPUT`" and "`READ`" Statement handlers. On entry register pair HL points to the first character of the text string to be converted. On exit register pair HL points to the character following the string, the numeric operand is in DAC and the type code in VALTYP. Examples of the three types are:

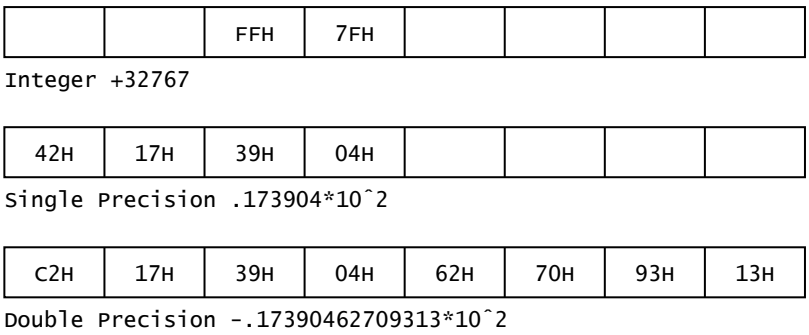


Figure 41: Numeric Types in DAC

An integer is a sixteen bit binary number in two's complement form, it is stored LSB first, MSB second at DAC+2. An integer can range from 8000H (-32768) to 7FFFH (+32767).

A floating point number consists of an exponent byte and a three or seven byte mantissa. The exponent is kept in signed binary form and can range from 01H (-63) through 40H (0) up to 7FH (+63), the special value of 00H is used for the number zero. These exponent values are for a normalized mantissa. The Interpreter presents exponent-form numbers to the user with a leading digit, this results in an asymmetric exponent range of E-64 to E+62. Bit 7 of the exponent byte holds the mantissa sign, 0 for positive and 1 for negative, the mantissa itself is kept in packed BCD form with two digits per byte. It should be noted that the Interpreter uses the contents of VALTYP to determine a number's type, not the format of the number itself.

Conversion starts by examining the first text character. If this is an "&" control transfers to the special radix conversion routine (4EB8H), if it is a leading sign character it is temporarily saved. Successive numeric characters are then taken and added to the integer product with appropriate multiplications by ten as each new digit is found. If the value of the product exceeds 32767, or a decimal point is found, the product is converted to single precision and any further characters placed directly in DAC.

If a seventh digit is found the product is changed to double precision, if more than fourteen digits are found the excess digits are read but ignored.

Conversion ceases when a non-numeric character is found. If this a type definition character ("%", "#" or "!") the appropriate conversion routine is called and control transfers to the exit point (331EH). If it is an exponent prefix ("E", "e", "D" or "d") one of the conversion routines will also be used and then the following digits converted to a binary exponent in register E. At the exit point (331EH) the product's type is checked via the GETYPR standard routine. If it is single precision or double precision the exponent is calculated by first subtracting the fractional digit count, in register B, from the total digit count, in register D, to produce the leading digit count. This is then added to any explicitly stated exponent, in register E, and placed at DAC+0 as the exponent.

The leading sign character is restored and the product negated if required (2E86H), if the product is integer the routine then terminates. If the product is single precision control terminates by checking for the special value of -32768 (2FA2H). If the product is double precision control terminates by rounding up from the fifteenth digit (273CH).

```
Address... 340AH
```

This routine is used by the error handler to display the message " in " (6678H) followed by the line number supplied in register pair HL (3412H).

```
Address... 3412H
```

This routine displays the unsigned binary integer supplied in register pair HL. The operand is placed in DAC as an integer (2F99H), converted to text (3441H) and then displayed (6677H).

```
Address... 3425H
```

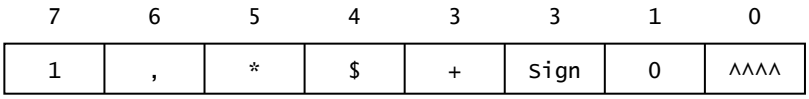
This routine converts the numeric operand contained in DAC to textual form in FBUFFR. The address of the first character of the resulting text is returned in register pair HL, the text is terminated by a zero byte. The operand is first converted to double precision (375FH). The BCD digits of the mantissa are then unpacked, converted to ASCII and placed in FBUFFR (36B3H). The position of the decimal point is determined by the exponent, for example:

```
.999*10 ^ +2 = 99.9
.999*10 ^ +1 = 9.99
.999*10 ^ +0 = .999
.999*10 ^ -1 = .0999
```

If the exponent is outside the range 10^-1 to 10^14 the number is presented in exponential form. In this case the decimal point is placed after the first digit and the exponent is converted from binary

and follows the mantissa.

An alternative entry point to the routine exists at 3426H for the "`PRINT USING`" statement handler. With this entry point the number of characters to prefix the decimal point is supplied in register B, the number of characters to point fix it in register C and a format byte in register A:



7	6	5	4	3	3	1	0
1	,	*	$	+	Sign	0	^^^^

Figure 42: Format Byte

Operation in this mode is fairly similar to the normal mode but with the addition of extra facilities. Once the operand has been converted to double precision the exponential form will be assumed if bit 0 of the format byte is set. The mantissa is shifted to the right in DAC and rounded up to lose unwanted postfix digits (377BH). As the mantissa is converted to ASCII (36B3H) commas will be inserted at the appropriate points if bit 6 of the format byte is set. During post-conversion formatting (351CH) unused prefix positions will be filled with asterisks if bit 5 is set, a pound prefix may be added by setting bit 4. Bit 3 enables the "+" sign for positive numbers if set, otherwise a space is used. Bit 2 places any sign at the front if reset and at the back if set.

The entry point to the routine at 3441H is used to convert unsigned integers, notably line numbers, to their textual form. For example 9000H, when treated as a normal integer, would be converted to -28672. By using this entry point 36864 would be produced instead. The operand is converted by successive division with the factors 10000, 1000, 100, 10 and 1 and the resulting digits placed in FBUFFR (36DBH).

```
Address... 3710H
```

This table contains the five constants used by the numeric output routine: 10000, 1000, 100, 10, 1.

```
Address... 371AH
```

This routine is used by the "`BIN`$" function to convert a numeric operand contained in DAC to textual form. Register B is loaded with the group size (1) and control transfers to the general conversion routine (3724H).

```
Address... 371EH
```

This routine is used by the "`OCT`$" function to convert a numeric operand contained in DAC to textual form. Register B is loaded with the group size (3) and control transfers to the general conversion routine (3724H).

```
Address... 3722H
```

This routine is used by the "`HEX`$" function to convert a numeric operand contained in DAC to textual form. Register B is loaded with the group size (4) and the operand converted to a binary integer in register pair HL (5439H). Successive groups of 1, 3 or 4 bits are shifted rightwards out of the operand, converted to ASCII digits and placed in FBUFFR. When the operand is all zeroes the routine terminates with the address of the first text character in register pair HL, the string is terminated with a zero byte.

```
Address... 3752H
```

This routine is used during numeric output to return an operand's digit count in register B and the address of its least significant byte in register pair HL. For single precision B=6 and HL=DAC+3, for double precision B=14 and HL=DAC+7.

```
Address... 375FH
```

This routine is used during numeric output to convert the numeric operand in DAC to double precision (303AH).

```
Address... 377BH
```

This routine is used during numeric output to shift the mantissa in DAC rightwards (27DBH), the inverse of the digit count is supplied in register A. The result is then rounded up from the fifteenth digit (2741H).

```
Address... 37A2H
```

This routine is used during numeric output to return the inverse of the fractional digit count in a floating point operand. This is computed by subtracting the exponent from the operand's digit count (6 or 14).

```
Address... 37B4H
```

This routine is used during numeric output to locate the last non-zero digit of the mantissa contained in DAC. Its address is returned in register pair HL.

```
Address... 37C8H
```

This routine is used by the Expression Evaluator to exponentiate (^) two single precision operands. The first operand is contained in registers C, B, E, D and the second in DAC, the result is returned in DAC. The first operand is copied to ARG (3280H), pushed onto the stack (2CC7H) and exchanged with DAC (2C6FH). The second operand is then popped into ARG and control drops into the double precision exponentiation routine.

```
Address... 37D7H
```

This routine is used by the Expression Evaluator to exponentiate (^) two double precision operands. The first operand is contained in DAC and the second in ARG, the result is returned in DAC. The result is usually computed using:

```
X^P=EXP(P*LOG(X))
```

An alternative, much faster, method is possible if the power operand is an integer. This is tested for by extracting the integer part of the operand and comparing for equality with the original value (391AH). A positive result to this test means that the faster method can be used, this is described below.

```
Address... 383FH
```

This routine is used by the Expression Evaluator to exponentiate (^) two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. The routine operates by breaking the problem down into simple multiplications:

```
6^13=6^1101=(6^8)*(6^4)*(6^1)
```

As the power operand is in binary form a simple right shift is sufficient to determine whether a particular intermediate product needs to be included in the result. The intermediate products themselves are obtained by cumulative multiplication of the operand each time the computation loop is traversed. If the product overflows at any time it is converted to single precision. Upon completion the power operand is checked, if it is negative the product is reciprocated as X^-P=1/X^P.

```
Address... 390DH
```

This routine is used during exponentiation to multiply two integers (3193H), it returns Flag NZ if the result has overflowed to single precision.

```
Address... 391AH
```

This routine is used during exponentiation to check whether a double precision power operand consists only of an integer part, if so it returns Flag NC.

```
Address... 392EH
```

This table of addresses is used by the Interpreter Runloop to find the handler for a statement token. Although not part of the table the associated keywords are included below:

TO	STATEMENT	TO	SATEMENT	TO	STMT
63EAH	END	00C3H	CLS	5B11H	CIRCLE
4524H	FOR	51C9H	WIDTH	7980H	COLOR
6527H	NEXT	485DH	ELSE	5D6EH	DRAW
485BH	DATA	6438H	TRON	59C5H	PAINT
4B6CH	INPUT	6439H	TROFF	00C0H	BEEP
5E9FH	DIM	643EH	SWAP	73E5H	PLAY
4B9FH	READ	6477H	ERASE	57EAH	PSET
4880H	LET	49AAH	ERROR	57E5H	PRESET
47E8H	GOTO	495DH	RESUME	73CAH	SOUND
479EH	RUN	53E2H	DELETE	79CCH	SCREEN
49E5H	IF	49B5H	AUTO	7BE2H	VPOKE
63C9H	RESTORE	5468H	RENUM	7A48H	SPRITE
47B2H	GOSUB	4718H	DEFSTR	7B37H	VDP
4821H	RETURN	471BH	DEFINT	7B5AH	BASE
485DH	REM	471EH	DEFSNG	55A8H	CALL
63E3H	STOP	4721H	DEFDBL	7911H	TIME
4A24H	PRINT	4B0EH	LINE	786CH	KEY
64AFH	CLEAR	6AB7H	OPEN	7E4BH	MAX
522EH	LIST	7C52H	FIELD	73B7H	MOTOR
6286H	NEW	775BH	GET	6EC6H	BLOAD
48E4H	ON	7758H	PUT	6E92H	BSAVE

TO	STATEMENT	TO	SATEMENT	TO	STMT
401CH	WAIT	6C14H	CLOSE	7C16H	DSKO$
501DH	DEF	6B5DH	LOAD	7C1BH	SET
5423H	POKE	6B5EH	MERGE	7C20H	NAME
6424H	CONT	6C2FH	FILES	7C25H	KILL
6FB7H	CSAVE	7C48H	LSET	7C2AH	IPL
703FH	CLOAD	7C4DH	RSET	7C2FH	COPY
4016H	OUT	6BA3H	SAVE	7C34H	CMD
4A1DH	LPRINT	6C2AH	LFILES	7766H	LOCATE
5229H	LLIST				

```
Address... 39DEH
```

This table of addresses is used by the Factor Evaluator to find the handler for a function token. Although not part of the table the associated keywords are included with the addresses shown below:

TO	FUNCTION	TO	FUNCTION	TO	FUNCTION
6861H	LEFT$	4FCCH	POS	30BEH	FIX
6891H	RIGHT$	67FFH	LEN	7940H	STICK
689AH	MID$	6604H	TR$	794CH	TRIG
2E97H	SGN	68BBH	VAL	795AH	PDL
30CFH	INT	680BH	ASC	7969H	PAD
2E82H	ABS	681BH	CHR$	7C39H	DSKF
2AFFH	SQR	541CH	PEEK	6D39H	FPOS
2BDFH	RND	7BF5H	VPEEK	7C66H	CVI
29ACH	SIN	6848H	SPACE$	7C6BH	CVS
2A72H	LOG	7C70H	OCT$	7C70H	CVD
2B4AH	EXP	65FAH	HEX$	6D25H	EOF
2993H	COS	4FC7H	LPOS	6D03H	LOC
29FBH	TAN	6FFFH	BIN$	6D14H	LOF

TO	FUNCTION	TO	FUNCTION	TO	FUNCTION
2A14H	ATN	2F8AH	CINT	7C57H	MKI$
69F2H	FRE	2FB2H	CSNG	7C5CH	MKS$
4001H	INP	303AH	CDBL	7C61H	MKD$

```
Address... 3A3EH
```

This table of addresses is used during program tokenization as an index into the BASIC keyword table (3A72H). Each of the twenty six entries defines the starting address of one of the keyword sub-blocks. The first entry points to the keywords beginning with the letter "A", the second to those beginning with the letter "B" and so on.

```
3A72H ... A   3B9FH ... J    3C8EH ... S
3A88H ... B   3BA0H ... K    3CDBH ... T
3A9FH ... C   3BA8H ... L    3CF6H ... U
3AF3H ... D   3BE8H ... M    3CFFH ... V
3B2EH ... E   3C09H ... N    3D16H ... W
3B4FH ... F   3C18H ... O    3D20H ... X
3B69H ... G   3C2BH ... P    3D24H ... Y
3B7BH ... H   3C5DH ... Q    3D25H ... Z
3B80H ... I   3C5EH ... R
```

```
Address... 3A72H
```

This table contains the BASIC keywords and tokens. Each of the twenty-six blocks within the table contains all the keywords beginning with a particular letter, it is terminated with a zero byte. Each keyword is stored in plain text with bit 7 set to mark the last character, this is followed immediately by the associated token. The first character of the keyword need not be stored as this is implied by its position in the table' The keywords and tokens are listed below in full, note that the "J", "Q", "Y" and "Z" blocks are empty:

```
AUTO   A9H  DSKF   26H  LIST    93H  REM      8FH
AND    F6H  DRAW   BEH  LFILES  BBH  RESUME   A7H
ABS    06H  ELSE   A1H  LOG     0AH  RSET     B9H
ATN    0EH  END    81H  LOC     2CH  RIGHT$   02H
ASC    15H  ERASE  A5H  LEN     12H  RND      08H
ATTR$  E9H  ERROR  A6H  LEFT$   01H  RENUM    AAH
BASE   C9H  ERL    E1H  LOF     2DH  SCREEN   C5H
BSAVE  D0H  ERR    E2H  MOTOR   CEH  SPRITE   C7H
BLOAD  CFH  EXP    0BH  MERGE   B6H  STOP     90H
BEEP   C0H  EOF    2BH  MOD     FBH  SWAP     A4H
BIN$   1DH  EQV    F9H  MKI$    2EH  SET      D2H
```

```
CALL   CAH  FOR    82H  MKS$    2FH  SAVE    BAH
CLOSE  B4H  FIELD  B1H  MKD$    30H  SPC(    DFH
COPY   D6H  FILES  B7H  MID$    03H  STEP    DCH
CONT   99H  FN     DEH  MAX     CDH  SGN     04H
CLEAR  92H  FRE    0FH  NEXT    83H  SQR     07H
CLOAD  9BH  FIX    21H  NAME    D3H  SIN     09H
CSAVE  9AH  FPOS   27H  NEW     94H  STR$    13H
CSRLIN E8H  GOTO   89H  NOT     E0H  STRING$ E3H
CINT   1EH  GO TO  89H  OPEN    B0H  SPACE$  19H
CSNG   1FH  GOSUB  8DH  OUT     9CH  SOUND   C4H
CDBL   20H  GET    B2H  ON      95H  STICK   22H
CVI    28H  HEX$   1BH  OR      F7H  STRIG   23H
CVS    29H  INPUT  85H  OCT$    1AH  THEN    DAH
CVD    2AH  IF     8BH  OFF     EBH  TRON    A2H
COS    0CH  INSTR  E5H  PRINT   91H  TROFF   A3H
CHR$   16H  INT    05H  PUT     B3H  TAB(    DBH
CIRCLE BCH  INP    10H  POKE    98H  TO      D9H
COLOR  BDH  IMP    FAH  POS     11H  TIME    CBH
CLS    9FH  INKEY$ ECH  PEEK    17H  TAN     0DH
CMD    D7H  IPL    D5H  PSET    C2H  USING   E4H
DELETE A8H  KILL   D4H  PRESET  C3H  USR     DDH
DATA   84H  KEY    CCH  POINT   EDH  VAL     14H
DIM    86H  LPRINT 9DH  PAINT   BFH  VARPTR  E7H
DEFSTR ABH  LLIST  9EH  PDL     24H  VDP     C8H
DEFINT ACH  LPOS   1CH  PAD     25H  VPOKE   C6H
DEFSNG ADH  LET    88H  PLAY    C1H  VPEEK   18H
DEFDBL AEH  LOCATE D8H  RETURN  8EH  WIDTH   A0H
DSKO$  D1H  LINE   AFH  READ    87H  WAIT    96H
DEF    97H  LOAD   B5H  RUN     8AH  XOR     F8H
DSKI$  EAH  LSET   B8H  RESTORE 8CH
```

```
Address... 3D26H
```

This twenty-one byte table is used by the Interpreter during program tokenization. It contains the ten single character keywords and their tokens:

```
+ ... F1H    * ... F3H   ^ ... F5H    ' ... E6H = ... EFH
- ... F2H    / ... F4H   \ ... FCH    > ... EEH < ... F0H
```

```
Address... 3D3BH
```

This table is used by the Expression Evaluator to determine the precedence level for a given infix operator, the higher the table value the greater the operator's precedence. Not included are the precedences for the relational operators (64H), the "`NOT`" operator (5AH) and the negation operator (7DH), these are defined directly by the Expression and Factor Evaluators.

```
79H ... +       46H ... OR
79H ... -       3CH ... XOR
7CH ... *       32H ... EQV
7CH ... /       28H ... IMP
7FH ... ^       7AH ... MOD
50H ... AND     7BH ... \
```

```
Address... 3D47H
```

This table is used to convert the result of a user defined function to the same type as the Variable used in the function definition. It contains the addresses of the type conversion routines:

```
303AH ... CDBL
0000H ... Not used
2F8AH ... CINT
3058H ... Check string type
2FB2H ... CSNG
```

```
Address... 3D51H
```

This table of addresses is used by the Expression Evaluator to find the handler for a particular infix math operator when both operands are double precision:

```
269AH ... +
268CH ... -
27E6H ... *
289FH ... /
37D7H ... ^
2F83H ... Relation
```

```
Address... 3D5DH
```

This table of addresses is used by the Expression Evaluator to find the handler for a particular infix math operator when both operands are single precision:

```
324EH ... +
3257H ... -
325CH ... *
3267H ... /
37C8H ... ^
2F21H ... Relation
```

```
Address... 3D69H
```

This table of addresses is used by the Expression Evaluator to find the handler for a particular infix math operator when both operands are integer:

```
3172H ... +
3167H ... -
3193H ... *
4DB8H ... /
383FH ... ^
2F4DH ... Relation
```

```
Address... 3D75H
```

This table contains the Interpreter error messages, each one is stored in plain text with a zero byte terminator. The associated error codes are shown below for reference only, they do not form part of the table:

```
01 NEXT without FOR             19 Device I/O error
02 Syntax error                 20 Verify error
03 RETURN without GOSUB         21 No RESUME
04 Out of DATA                  22 RESUME without error
05 Illegal function call        23 Unprintable error
06 Overflow                     24 Missing operand
07 Out of memory                25 Line buffer overflow
08 Undefined line number        50 FIELD overflow
09 Subscript out of range       51 Internal error
10 Redimensioned array          52 Bad file number
11 Division by zero             53 File not found
12 Illegal direct               54 File already open
13 Type mismatch                55 Input past end
14 Out of string space          56 Bad file name
15 String too long              57 Direct statement in file
16 String formula too complex   58 Sequential I/O only
17 Can't CONTINUE               59 File not OPEN
18 Undefined user function
```

```
Address... 3FD2H
```

This is the plain text message "`in`" terminated by a zero byte.

```
Address... 3FD7H
```

This is the plain text message " `Ok` ", CR, LF terminated by a zero byte.

```
Address... 3FDCH
```

This is the plain text message " `Break` " terminated by a zero byte.

```
Address... 3FE2H
```

This routine searches the Z80 stack for the " `FOR` " loop parameter block whose loop Variable address is supplied in register pair DE. The search is started four bytes above the current Z80 SP to allow for the caller's return address and the Runloop return address. If no " `FOR` " token (82H) exists the routine terminates Flag NZ, if one is found the loop Variable address is taken from the parameter block and checked. The routine terminates Flag Z upon a successful match with register pair HL pointing to the type byte of the parameter block. Otherwise the search moves up twenty-two bytes to the start of the next parameter block.

```
Address... 4001H
```

This routine is used by the Factor Evaluator to apply the " `INP` " function to an operand contained in DAC. The port number is checked (5439H), the port read and the result placed in DAC as an integer (4FCFH).

```
Address... 400BH
```

This routine first evaluates an operand in the range -32768 to +65535 (542FH) and places it in register pair BC. After checking for a comma, via the SYNCHR standard routine, it evaluates a second operand in the range 0 to 255 (521CH) and places this in register A.

```
Address... 4016H
```

This is the " `OUT` " statement handler. The port number and data byte are evaluated (400BH) and the data byte written to the relevant Z80 port.

```
Address... 401CH
```

This is the " `WAIT` " statement handler. The port number and " `AND` " operands are first evaluated (400BH) followed by the optional " `XOR` " operand (521CH). The port is then repeatedly read and the operands applied, XOR then AND, until a non-zero result is obtained. Contrary to the information given in some MSX manuals the loop can be broken by the CTRL-STOP key as the CKCNTC standard routine is called from inside it.

```
Address... 4039H
```

This routine is used by the Runloop when it encounters the end of the program text while in program mode. ONEFLAG is checked to see whether it still contains an active error code. If so a "`No RESUME`" error is generated, otherwise program termination continues normally (6401H). The idea behind this routine is to catch any "`ON ERROR`" handlers without a "`RESUME`" statement at the end.

```
Address... 404FH
```

This routine is used by the "`READ`" statement handler when an error is found in a "`DATA`" statement. The line number contained in DATLIN is copied to CURLIN so the error handler will flag the "`DATA`" line as the illegal statement rather than the program line. Control then drops into the "`Syntax error`" generator.

```
Address... 4055H
```

This is a group of nine error generators, register E is loaded with the relevant error code and control drops into the error handler:

ADDRESS	ERROR
4055H	Syntax error
4058H	Division by zero
405BH	NEXT without FOR
405EH	Redimensioned array
4061H	Undefined user function
4064H	RESUME without error
4067H	Overflow error
406AH	Missing operand
406DH	Type mismatch

```
Address... 406FH
```

This is the Interpreter error handler, all error generators transfer to here with an error code in register E. VLZADR is first checked to see if the "`VAL`" statement handler has changed the program text, if so the original character is restored from VLZDAT. The current line number is then copied from CURLIN to ERRLIN and DOT and the Z80 stack is restored from SAVSTK (62F0H). The error code is placed in

ERRFLG, for use by the "`ERR`" function, and the current program text position copied from SAVTXT to ERRTXT for use by the "`RESUME`" statement handler. The error line number and program text position are also copied to OLDLIN and OLDTXT for use by the "`CONT`" statement handler. ONELIN is then checked to see if a previous "`ON ERROR`" statement has been executed. If so, and providing no error code is already active, control transfers to the Runloop (4620H) to execute the BASIC error recovery statements.

Otherwise the error code is used to count through the error message table at 3D75H until the required one is reached. A CR,LF is issued (7323H) and the screen forced back to text mode via the TOTEXT standard routine. A BELL code is then issued and the error message displayed (6678H). Assuming the Interpreter is in program mode, rather than direct mode, this is followed by the line number (340AH) and control drops into the "`OK`" point.

```
Address... 411FH
```

This is the re-entry point to the Interpreter Mainloop for a terminating program. The screen is forced to text mode via the TOTEXT standard routine, the printer is cleared (7304H) and I/O buffer 0 closed (6D7BH). A CR,LF is then issued to the screen (7323H), the message "`OK`" is displayed (6678H) and control drops into the Mainloop.

```
Address... 4134H
```

This is the Interpreter Mainloop. CURLIN is first set to FFFFH to indicate direct mode and AUTFLG checked to see if "`AUTO`" mode is on. If so the next line number is taken from AUTLIN and displayed (3412H). The Program Text Area is then searched to see if this line already exists (4295H) and either an asterisk or space displayed accordingly.

The ISFLIO standard routine is then used to determine whether a "`LOAD`" statement is active. If so the program line is collected from the cassette (7374H), otherwise it is taken from the console via the PINLIN standard routine. If the line is empty or the CTRL-STOP key has been pressed control transfers back to the start of the Mainloop (4134H) with no further action. If the line commences with a line number this is converted to an unsigned integer in register pair DE (4769H). The line is then converted to tokenized form and placed in KBUF (42B2H). If no line number was found at the start of the line control then transfers to the Runloop (6D48H) to execute the statement.

Assuming the line commences with a line number it is tested to see if it is otherwise empty and the result temporarily saved. The line number is copied to DOT and AUTLIN increased by the contents of AUTINC, if AUTLIN now exceeds 65530 the "`AUTO`" mode is turned off. The Program Text Area is then searched (4295H) to find a matching line number or, failing this, the position of the next highest line number. If no matching line number is found and the line is empty and "`AUTO`" mode is off an "`Undefined line number`" error is generated (481CH). If a matching line number is found and the line is empty and "`AUTO`" mode is on the Mainloop simply skips to the next statement (4237H).

Otherwise any pointers in the Program Text Area are converted back to line numbers (54EAH) and any existing program line deleted (5405H). Assuming the new program line is non-empty the Program Text Area is opened up by the required amount (6250H) and the tokenized program line copied from KBUF.

The Program Text Area links are then recalculated (4257H), the Variable Storage Areas are cleared (629AH) and control transfers back to the start of the Mainloop.

```
Address... 4253H
```

This routine recalculates the Program Text Area links after a program modification. The first two bytes of each program line contain the starting address of the following line, this is called the link. Although the link increases the amount of storage required per program line it greatly reduces the time required by the Interpreter to locate a given line.

An example of a typical program line is shown below, in this case the line "`10 PRINT 9`" situated at the start of the Program Text Area (8001H):

09H	80H	0AH	00H	91H	20H	1AH	00H

Figure 43: Program Line

In the above example the link is stored in Z80 word order (LSB,MSB) and is immediately followed by the binary line number, also in word order. The statement itself is composed of a "`PRINT`" token (91H), a single space, the number nine and the end of line character (00H). Further details of the storage format can be found in the tokenizing routine (42B2H).

Each link is recalculated simply by scanning through the line until the end of line character is found. The process is complete when the end of the Program Storage Area, marked by the special link of 0000H, is reached.

```
Address... 4279H
```

This routine is used by the "`LIST`" statement handler to collect up to two line number operands from the program text. If the first line number is present it is converted to an unsigned integer in register pair DE (475FH), if not a default value of 0000H is returned. If the second line number is present it must be preceded by a "-" token (F2H) and is returned on the Z80 stack, if not a default value of 65530 is returned. Control then drops into the program text search routine to find the first referenced program line.

```
Address... 4295H
```

This routine searches the Program Text Area for the program line whose line number is supplied in register pair DE. Starting at the address contained in TXTTAB each program line is examined for a match. If an equal line number is found the routine terminates with Flag Z,C and register pair BC pointing to the start of the program line. If a higher line number is found the routine terminates Flag NZ,NC and if the end link is reached the routine terminates Flag Z,NC.

```
Address... 42B2H
```

This routine is used by the Interpreter Mainloop to tokenize a line of text. On entry register pair HL points to the first text character in BUF. On exit the tokenized line is in KBUF, register pair BC holds its length and register pair HL points to its start.

Except after opening quotes or after the " `REM` ", " `CALL` " or " `DATA` " keywords any string of characters matching a keyword is replaced by that keyword's token. Lower case alphabetics are changed to upper case for keyword comparison. The character " `?` " is replaced by the " `PRINT` " token (91H) and the character "'" by ":" (3AH), " `REM` " token (8FH), "'" token (E6H). The " `ELSE` " token (A1H) is preceded by a statement separator (3AH). Any other miscellaneous characters in the text are copied without alteration except that lower case alphabetics are converted to upper case. Those tokens smaller than 80H, the function tokens, cannot be stored directly in KBUF as they will conflict with ordinary text. Instead the sequence FFH, token+80H is used.

Numeric constants are first converted into one of the standard types in DAC (3299H). They are then stored in one of several ways depending upon their type and magnitude, the general idea being to minimize memory usage:

```
0BH LSB MSB ................... Octal number
0CH LSB MSB ................... Hex number
11H to 1AH .................... Integer 0 to 9
0FH LSB ....................... Integer 10 to 255
1CH LSB MSB ................... Integer 256 to 32767
1DH EE DD DD DD ............... Single Precision
1FH EE DD DD DD DD DD DD DD ... Double Precision
```

There is no specific token for binary numbers, these are left as character strings. This would appear to be a legacy from earlier versions of Microsoft BASIC. Any sign prefixing a number is regarded as an operator and is stored as a separate token, negative numbers are not produced during tokenization. As double precision numbers occupy so much space a line containing too many, for example PRINT 1#,1#,1# etc. may cause KBUF to fill up. If this happens a " `Line buffer overflow` " error is generated.

Any number following one of the keyword tokens in the table at 43B5H is considered to be a line number operand and is stored with a different token:

```
0DH LSB MSB ................... Pointer
0EH LSB MSB ................... Line number
```

During tokenization only the normal type (0EH) is generated, when a program actually runs these line number operands are converted to the address pointer type (0DH).

```
Address... 43B5H
```

This table of tokens is used during tokenization to check for the keywords which take line number operands. The keywords themselves are listed below:

```
RESTORE     RUN
AUTO        LIST
RENUM       LLIST
DELETE      GOTO
RESUME      RETURN
ERL         THEN
ELSE        GOSUB
```

```
Address... 4524H
```

This is the "`FOR`" statement handler. The loop Variable is first located and assigned its initial value by the "`LET`" handler (4880H), the address of the loop Variable is returned in register pair DE. The end of the statement is found (485BH) and its address placed in ENDFOR. The Z80 stack is then searched (3FE6H) for any parameter blocks using the same loop Variable. For each one found the current ENDFOR address is compared with that of the parameter block, if there is a match that section of the stack is discarded. This is done in case there are any incomplete loops as a result of a "`GOTO`" back to the "`FOR`" statement from inside the loop.

The termination operand and optional "`STEP`" operand are then evaluated and converted to the same type as the loop Variable. After checking that stack space is available (625EH) a twenty-five byte parameter block is pushed onto the Z80 stack. This is made up of the following:

```
2 bytes ... ENDFOR address
2 bytes ... Current line number
8 bytes ... Loop termination value
8 bytes ... STEP value
1 byte  ... Loop type
1 byte  ... STEP direction
2 bytes ... Address of loop Variable
1 byte  ... FOR token (82H)
```

The parameter block remains on the stack for use by the "`NEXT`" statement handler until termination is reached, it is then discarded. The size of the block remains constant even though, for integer and single precision loop Variables, the full eight bytes are not required for the termination and STEP values. In these cases the least significant bytes are packed out with garbage.

It should be noted that the type of arithmetic operation performed by the " `NEXT` " statement handler, and hence the loop execution speed, depends entirely upon the loop Variable type and not the operand types. For the fastest program execution integer type Variables, N% for example, should be used.

```
Address... 4601H
```

This is the Runloop, each statement handler returns here upon completion so the Interpreter can proceed to the next statement. The current Z80 SP is copied to SAVSTK for error recovery purposes and the CTRL-STOP key checked via the ISCNTC standard routine. Any pending interrupts are processed (6389H) and the current program text position, held in register pair HL throughout the Interpreter, is copied to SAVTXT.

The current program character is then examined, if this is a statement separator (3AH) control transfers immediately to the execution point (4640H). If it is anything else but an end of line character (00H) a " `Syntax error` " is generated (4055H) as there is spurious text at the end of the statement. Register pair HL is advanced to the first character of the new program line and the link examined, if this is zero the program is terminated (4039H). Otherwise the line number is taken from the new line and placed in CURLIN. If TRCFLG is non-zero the line number is displayed (3412H) enclosed by square brackets, control then drops into the execution point.

```
Address... 4640H
```

This is the Runloop execution point. A return to the start of the Runloop (4601H) is pushed onto the Z80 stack and the first character taken from the new statement via the CHRGTR standard routine. If it is an underline character (5FH) control transfers to the " `CALL` " statement handler (55A7H). If it is smaller than 81H, the smallest statement token, control transfers to the " `LET` " handler (4880H). If it is larger than D8H, the largest statement token, it is checked to see if it is one of the function tokens allowed as a statement (51ADH). Otherwise the handler address is taken from the table at 392EH and pushed onto the stack. Control then drops into the CHRGTR standard routine to fetch the next program character before control transfers to the statement handler.

```
Address... 4666H
Name..... CHRGTR
Entry..... HL points to current program character
Exit..... A=Next program character
Modifies.. AF, HL
```

Standard routine to fetch the next character from the program text. Register pair HL is incremented and the character placed in register A. If it is a space, TAB code (09H) or LF code (0AH) it is skipped over. If it is a statement separator (3AH) or end of line character (00H) the routine terminates with Flag Z,NC. If it is a digit from "0" to "9" the routine terminates with Flag NZ,C. If it is any other character apart from the numeric prefix tokens the routine terminates Flag NZ,NC. If the character is

one of the numeric prefix tokens then it is placed in CONSAV and the operand copied to CONLO. The type code is placed in CONTYP and the address of the trailing program character in CONTXT.

```
Address... 46E8H
```

This routine is used by the Factor Evaluator and during detokenization to recover a numeric operand when one of the prefix tokens is returned by the CHRGTR standard routine. The prefix token is first taken from CONSAV, if it is anything but a line number or pointer token the operand is copied from CONLO to DAC and the type code copied from CONTYP to VALTYP. If it is a line number it is converted to single precision and placed in DAC (3236H). If it is a pointer the original line number is recovered from the referenced program line, converted to single precision and placed in DAC (3236H).

```
Address... 4718H
```

This is the "`DEFSTR`" statement handler. Register E is loaded with the string type code (03H) and control drops into the general type definition routine.

```
Address... 471BH
```

This is the "`DEFINT`" statement handler. Register E is loaded with the integer type code (02H) and control drops into the general type definition routine.

```
Address... 471EH
```

This is the "`DEFSNG`" statement handler. Register E is loaded with the single precision type code (04H) and control drops into the general type definition routine.

```
Address... 4721H
```

This is the "`DEFDBL`" statement handler. Register E is loaded with the double precision type code (08H) and the first range definition character checked (64A7H). If this is not upper case alphabetic a "`Syntax error`" is generated (4055H). If a "-" token (F2H) follows the second range definition character is taken and checked (64A7H), the difference between the two determines the number of entries in DEFTBL that are filled with the type code.

```
Address... 4755H
```

This routine evaluates an operand and converts it to an integer in register pair DE (520FH). If the operand is negative an "`Illegal function call`" error is generated.

```
Address... 475FH
```

This routine is used by the statement handlers shown in the table at 43B5H to collect a single line number operand from the program text and convert it to an unsigned integer in register pair DE. If the first character in the text is a "." (2EH) the routine terminates with the contents of DOT. If it is one of the line number tokens (0DH or 0EH) the routine terminates with the contents of CONLO. Otherwise successive digits are taken and added to the product, with appropriate multiplications by ten, until a non-numeric character is found.

```
Address... 479EH
```

This is the "`RUN`" statement handler. If no line number operand is present in the program text the system is cleared (629AH) and control returns to the Runloop with register pair HL pointing to the start of the Program Storage Area. If a line number operand is present the system is cleared (62A1H) and control transfers to the "`GOTO`" statement handler (47E7H). Otherwise a following filename is assumed, for example `RUN "CAS:FILE"`, and control transfers to the "`LOAD`" statement handler (6B5BH);

```
Address... 47B2H
```

This is the "`GOSUB`" statement handler. After checking that stack space is available (625EH) the line number operand is collected and placed in register pair DE (4769H). The seven byte parameter block is then pushed onto the stack and control transfers to the "`GOTO`" handler (47EBH). The parameter block is made up of the following:

```
2 bytes ... End of statement address
2 bytes ... Current line number
2 bytes ... 0000H
1 byte  ... GOSUB token (8DH)
```

The parameter block remains on the stack until a "`RETURN`" statement is executed. It is then used to determine the original program text position after which it is discarded.

```
Address... 47CFH
```

This routine is used by the Runloop interrupt processor (6389H) to create a "`GOSUB`" type parameter block on the Z80 stack. An interrupt block is identical to a normal block except that the two zero bytes shown above are replaced by the address of the device's entry in TRPTBL. This address will be used by the "`RETURN`" statement handler to update the device's interrupt status once a subroutine has terminated. After pushing the parameter block control transfers to the Runloop to execute the program line whose address is supplied in register pair DE.

```
Address... 47E8H
```

This is the "`GOTO`" statement handler. The line number operand is collected (4769H) and placed in register pair HL. If it is a pointer control transfers immediately to the Runloop to begin execution at the new program text position. Otherwise the line number is compared with the current line number to determine the starting position for the program text search. If it is greater the search starts from the end of this line (4298H), if it is smaller it starts from the beginning of the Program Text Area (4295H). If the referenced line cannot be found an "`Undefined line number`" error is generated (481CH). Otherwise the line number operand is replaced by the referenced program line's address and its token changed to the pointer type (5583H). Control then transfers to the Runloop to execute the referenced program line.

```
Address... 481CH
```

This is the "`Undefined line number`" error generator.

```
Address... 4821H
```

This is the "`RETURN`" statement handler. A dummy loop Variable address is placed in register pair DE and the Z80 stack searched (3FE2H) to find the first parameter block not belonging to a "`FOR`" loop, this section of stack is then discarded. If no "`GOSUB`" token (8DH) is found at this point a "`RETURN without GOSUB`" error is generated.

The next two bytes are then taken from the block, if they are non-zero the block was generated by an interrupt and the temporary "`STOP`" condition is removed (633EH). The program text is then examined, if anything follows the "`RETURN`" token itself it is assumed to be a line number operand and control transfers to the "`GOTO`" handler (47E8H). Otherwise the old line number and program text address are taken from the block and control returns to the Runloop.

```
Address... 485BH
```

This is the "`DATA`" statement handler. The program text is skipped over until a statement separator (3AH) or end of line character (00H) is found. This routine is also the "`REM`" and "`ELSE`" statement handler via the entry point at 485DH, in this case only the end of line character acts as a terminator.

```
Address... 4880H
```

This is the "`LET`" statement handler. The Variable is first located (5EA4H), its address saved in TEMP and the operand evaluated (4C64H). If necessary the operand's type is then changed to match that of the Variable (517AH). Assuming the operand is one of the three numeric types it is simply copied from DAC to the Variable in the Variable Storage Area (2EF3H). If the operand is a string type the

address of the string body is taken from the descriptor and checked. If it is in KBUF, as would be the case for an explicit string in a direct statement, the body is first copied to the String Storage Area and a new descriptor created (6611H). The descriptor is then freed from TEMPST (67EEH) and copied to the Variable in the Variable Storage Area (2EF3H).

```
Address... 48E4H
```

This is the "`ON ERROR`", "`ON DEVICE GOSUB`" and "`ON EXPRESSION`" statement handler. If the next program text character is not an "`ERROR`" token (A6H) control transfers to the "`ON DEVICE GOSUB`" and "`ON EXPRESSION`" handler (490DH). The program text is checked to ensure that a "`GOTO`" token (89H) follows and then the line number operand collected (4769H). The program text is searched to obtain the address of the referenced line (4293H) and this is placed in ONELIN. If the line number operand is non-zero the routine then terminates. If the line number operand is zero ONEFLG is checked to see if an error situation already exists (implying that the statement is inside a BASIC error recovery routine). If so control transfers to the error handler (4096H) to force an immediate error, otherwise the routine terminates normally.

```
Address... 490DH
```

This is the "`ON DEVICE GOSUB`" and "`ON EXPRESSION`" statement handler. If the next program text character is not a device token (7810H) control transfers to the "`ON EXPRESSION`" handler (4943H). After checking the program text for a "`GOSUB`" token (8DH) each of the line number operands required for a particular device is collected in turn (4769H). Assuming a given line number operand is non-zero the program text is searched to find the address of the referenced line (4293H) and this is placed in the device's entry in TRPTBL (785CH). The routine terminates when no more line number operands are found.

```
Address... 4943H
```

This is the "`ON EXPRESSION`" statement handler. The operand is evaluated (521CH) and the following "`GOSUB`" token (8DH) or "`GOTO`" token (89H) placed in register A. The operand is then used to count along the program text until register pair HL points to the required line number operand. Control then transfers back to the Runloop execution point (4646H) to decode the "`GOSUB`" or "`GOTO`" token.

```
Address... 495DH
```

This is the "`RESUME`" statement handler. ONEFLG is first checked to make sure that an error condition already exists, if not a "`RESUME without error`" is generated (4064H). If a non- zero line number operand follows control transfers to the "`GOTO`" handler (47EBH). If a "`NEXT`" token (83H) follows the position of the error is restored from ERRTXT and ERRLIN, the start of the next statement is found

(485BH) and the routine terminates. If there is no line number operand or if it is zero the position of the error is found from ERRTXT and ERRLIN and the routine terminates.

```
Address... 49AAH
```

This is the " `ERROR` " statement handler. The operand is evaluated and placed in register E (521CH). If it is zero an " `Illegal function call` " error is generated (475AH), otherwise control transfers to the error handler (406FH).

```
Address... 49B5H
```

This is the " `AUTO` " Statement handler. The optional start and increment line number operands, both with a default value of ten, are collected (475FH) and placed in AUTLIN and AUTINC. After making AUTFLG non-zero the Runloop return is destroyed and control transfers directly to the Mainloop (4134H).

```
Address... 49E5H
```

This is the " `IF` " statement handler. The operand is evaluated (4C64H) and, after checking for a " `GOTO` " token (89H) or " `THEN` " token (DAH), its sign is tested (2EA1H). If the operand is non- zero (true) the following text is executed either by an immediate transfer to the Runloop (4646H) or, for a line number operand, the " `GOTO` " handler (47E8H). If the operand is zero (false) the statement text is scanned (485BH) until an " `ELSE` " token (A1H) is found not balanced by an " `IF` " token (8BH) and execution re-commences.

```
Address... 4A1DH
```

This is the " `LPRINT` " statement handler. PRTFLG is set to 01H, to direct output to the printer, and control transfers to the " `PRINT` " handler (4A29H).

```
Address... 4A24H
```

This is the " `PRINT` " statement handler. The program text is first checked for a trailing buffer number and, if necessary, PTRFIL set to direct output to the required I/O buffer (6D57H). If no more program text exists a CR,LF is issued (7328H) and the routine terminates (4AFFH). Otherwise successive characters are taken from the program text and analyzed. If a " `USING` " token (E4H) is found control transfers to the " `PRINT USING` " handler (60B1H). If a ";" character is found control just transfers back to the start to fetch the next item (4A2EH). If a comma is found sufficient spaces are issued to bring the current print position, from TTYPOS, LPTPOS or an I/O buffer FCB, to an integral multiple of fourteen. If output is directed to the screen and the print position is equal to or greater than the contents of CLMLST or if output is directed to the printer and it is equal to or greater than 238 then a

CR,LF is issued instead (7328H). If a " `SPC(` " token (DFH) is found the operand is evaluated (521BH) and the required number of spaces are output. If a " `TAB(` " token (DBH) is found the operand is evaluated (521BH) and sufficient spaces issued to bring the current print position, from TTYPOS, LPTPOS or an I/O buffer FCB, to the required point.

If none of these characters is found the program text contains a data item which is then evaluated (4C64H). If the operand is a string it is simply displayed (667BH). If it is numeric it is first converted to text in FBUFFR (3425H) and a string descriptor created (6635H). If output is directed to an I/O buffer the resulting string is then displayed (667BH). If output is directed to the screen or printer the current print position, from TTYPOS or LPTPOS, is compared with the line length and a CR,LF issued (7328H) if the output will not fit on the line. The maximum line length is 255 for the printer and is taken from LINLEN for the screen. Once the string has been displayed control transfers back to the start of the handler.

```
Address... 4AFFH
```

This routine zeroes PRTFLG and PTRFIL to return the Interpreter's output to the screen.

```
Address... 4B0EH
```

This is the " `LINE INPUT` ", " `LINE INPUT#` " and " `LINE` " statement handler. If the following program text character is anything other than an " `INPUT` " token (85H) control transfers to the " `LINE` " statement handler (58A7H). If the following program text character is a "#" (23H) control transfers to the " `LINE INPUT#` " statement handler (6D8FH).

Any following prompt string is evaluated and displayed (4B7BH) and the Variable located (5EA4H) and checked to ensure that it is a string type (3058H). The line of text is collected from the console via the INLIN standard routine, if Flag C (CTRL-STOP) is returned control transfers to the " `STOP` " statement handler (63FEH). Otherwise the input string is analyzed and a descriptor created (6638H), control then transfers to the " `LET` " statement handler for assignment (4892H). It should be noted that the screen is not forced to text mode before the input is collected.

```
Address... 4B3AH
```

This is the plain text message " `?Redo from start` ", CR, LF terminated by a zero byte.

```
Address... 4B4DH
```

This routine is used by the " `READ/INPUT` " statement handler if it has failed to convert a data item to numeric form. If in " `READ` " mode (FLGINP is non-zero) a " `Syntax error` " is generated (404FH). Otherwise the message " `?Redo from start` " is displayed (6678H) and control returns to the statement handler.

```
Address... 4B62H
```

This is the " `INPUT#` " Statement handler. The buffer number is evaluated and PTRFIL set to direct input from the required I/O buffer (6D55H), control then transfers to the combined " `READ/INPUT` " statement handler (4B9BH).

```
Address... 4B6CH
```

This is the " `INPUT` " statement handler. If the next program text character is a "#" control transfers to the " `INPUT#` " statement handler (4B62H). Otherwise the screen is forced to text mode, via the TOTXT standard routine, and any prompt string analyzed (6636H) and displayed (667BH). A question mark is then displayed and a line of text collected from the console via the QINLIN standard routine. If this returns Flag C (CTRL-STOP) control transfers to the " `STOP` " handler (63FEH). If the first character in BUF is zero (null input) the handler terminates by skipping to the end of the statement (485AH), otherwise control drops into the combined " `READ/INPUT` " handler.

```
Address... 4B9FH
```

This is the " `READ` " statement handler, a large section is also used by the " `INPUT` " and " `INPUT#` " statements so the structure is rather awkward. Each Variable found in the program text is located in turn (5EA4H), for each one the corresponding data item is obtained and assigned to the Variable by the " `LET` " handler (4893H). When in " `READ` " mode the data items are taken from the program text using the initial contents of DATPTR (4C40H). When in " `INPUT` " or " `INPUT#` " mode the data items are taken from the text buffer BUF.

If the data items are exhausted in " `READ` " mode an " `Out of DATA` " error is generated. If they are exhausted in " `INPUT` " mode two question marks are displayed and another line fetched from the console via the QINLIN standard routine. If they are exhausted in " `INPUT#` " mode another line of text is copied to BUF from the relevant I/O buffer (6D83H). If the Variable list is exhausted while in " `INPUT` " mode the message " `Extra ignored` " is displayed (6678H) and the handler terminates (4AFFH). In " `INPUT#` " mode no message is displayed while in " `READ` " mode control terminates by updating DATPTR (63DEH). If a data item cannot be converted to numeric form (3299H) to match a numeric Variable control transfers to the " `?Redo from start` " routine (4B4DH).

```
Address... 4C2FH
```

The is the plain text message " `?Extra ignored` ", CR, LF terminated by a zero byte.

```
Address... 4C40H
```

This routine is used by the "`READ`" handler to locate the next "`DATA`" statement in the program text, the address to start from is supplied in register pair HL. Each program statement is examined until a "`DATA`" token (84H) is found whereupon the routine terminates (4BD1H). If the end link is reached an "`Out of DATA`" error is generated. As the search proceeds the line number of each program line is placed in DATLIN for use by the error handler.

```
Address... 4C5FH
```

This routine checks that the next character in the program text is the "=" token (EFH) and then drops into the Expression Evaluator. When entered at 4C62H it checks for "(".

```
Address... 4C64H
```

This is the Expression Evaluator. On entry register pair HL points to the first character of the expression to be evaluated. On exit register pair HL points to the character following the expression, the result is in DAC and the type code in VALTYP. For a string result the address of the string descriptor is returned at DAC+2. The descriptor itself comprising a single byte for the string length and two bytes for its address, will be in TEMPST or inside a string Variable.

An expression is a list of factors (4DC7H) linked together by operators with differing precedence levels. To process such an expression correctly the Expression Evaluator must be able to temporarily stack an intermediate result, if the next operator has a higher precedence than the current operator, and start afresh on a new calculation. It therefore has two basic operations, STACK and APPLY. For example:

```
3+250\2^2*3^3+1,

STACK:    3+        (\ follows)
STACK:    250\      (^ follows)
APPLY:    2^2=4     (* follows)
STACK:    4*        (^ follows)
APPLY:    3^3=27    (+ follows)
APPLY:    4*27=108  (+ follows)
APPLY:    250\108=2 (+ follows)
APPLY:    3+2=5     (+ follows)
APPLY:    5+1=6     (, follows)
```

Evaluation terminates when the next operator has a precedence equal to or lower than the initial precedence and the stack is empty. The expression delimiter, shown as a comma in the example, is regarded as having a precedence of zero and so will always halt evaluation. Normally the Expression Evaluator starts off with an initial precedence of zero but the entry point at 4C67H may be used to supply an alternative value in register D. This facility is used by the Factor Evaluator to restrict the range of evaluation when applying the monadic negation and "`NOT`" operators.

```
Address... 4D22H
```

This routine is used by the Expression Evaluator to apply an infix math operator (+-*/) to a pair of numeric operands. There are separate routines for the relational operators (4F57H) and the logical operators (4F78H). The first operand, its type code, and the operator token are supplied on the Z80 stack, the second operand and its type code are supplied in DAC and VALTYP. The types of both operands are first compared, if they differ the lowest precision operand is converted to match the higher. The operands are then moved to the positions required by the math routines. For integers the first operand is placed in register pair DE and the second in register pair HL. For single precision the first operand is placed in registers C, B, E, D and the second in DAC. For double precision the first operand is placed in DAC and the second in ARG. The operator token is then used to obtain the required address from the table at 3D51H, 3D5DH or 3D69H, depending upon the operand type, and control transfers to the relevant math routine.

```
Address... 4DB8H
```

This routine is used by the Expression Evaluator to divide two integer operands. The first operand is contained in register pair DE and the second in register pair HL, the result is returned in DAC. Both operands are converted to single precision (2FCBH) and control transfers to the single precision division routine (3265H).

```
Address... 4DC7H
```

This is the Factor Evaluator. On entry register pair HL points to the character before the factor to be evaluated. On exit register pair HL points to the character following the factor, the result is in DAC and the type code in VALTYP. A factor may be one of the following:

1. A numeric or string constant
2. A numeric or string Variable
3. A function
4. A monadic operator (+-NOT)
5. A parenthesized expression

The first character is taken from the program text via the CHRGTR standard routine and examined. If it is an end of Statement character a "`Missing operand`" error is generated (406AH). If it is an ASCII digit it is converted from textual form to one of the standard numeric types in DAC (3299H).

If it is upper case alphabetic (64A8H) it is a Variable and its current value is returned (4E9BH). If it is a numeric token the number is copied from CONLO to DAC (46B8H). If it is one of the FFH prefixed function tokens shown in the table at 39DEH it is decoded to transfer control to the relevant function handler (4ECFH). If it is the monadic "+" operator it is simply skipped over, only the monadic "-" operator (4E8DH) and monadic "`NOT`" operator (4F63H) require any action.

If it is an opening quote the following explicit string is analyzed and a descriptor created (6636H). If it is an "&" it is a non-decimal numeric constant and it is converted to one of the standard numeric types in DAC (4EB8H). If it is not one of the functions shown below then it must be a parenthesized expression (4E87H), otherwise a "`Syntax error`" is generated. The following function tokens are tested for directly and control transferred to the address shown:

```
ERR .... 4DFDH   ATTR$ .... 7C43H
ERL .... 4E0BH   VARPTR ... 4E41H
POINT .. 5803H   USR....... 4FD5H
TIME ... 7900H   INSTR .... 68EBH
SPRITE . 7A84H   INKEY$ ... 7347H
VDP .... 7B47H   STRING$ .. 6829H
BASE ... 7BCBH   INPUT$ ... 6C87H
PLAY ... 791BH   CSRLIN ... 790AH
DSKI$ .. 7C3EH   FN ....... 5040H
```

```
Address... 4DFDH
```

This routine is used by the Factor Evaluator to apply the "`ERR`" function. The contents of ERRFLG are placed in DAC as an integer (4FCFH).

```
Address... 4E0BH
```

This routine is used by the Factor Evaluator to apply the "`ERL`" function. The contents of ERRLIN are copied to DAC as a single precision number (3236H).

```
Address... 4E41H
```

This routine is used by the Factor Evaluator to apply the "`VARPTR`" function. If the function token is followed by a "#" the buffer number is evaluated (521BH), the I/O buffer FCB located (6A6DH) and its address placed in DAC as an integer (2F99H). Otherwise the Variable is located (5F5DH) and its address placed in DAC as an integer (2F99H).

```
Address... 4E8DH
```

This routine is used by the Factor Evaluator to apply the monadic "-" operator. Register D is set to a precedence value of 7DH, the factor evaluated (4C67H) and then negated (2E86H).

```
Address... 4E9BH
```

This routine is used by the Factor Evaluator to return the current value of a Variable. The Variable is first located (5EA4H). If it is a string Variable its address is placed in DAC to point to the descriptor. Otherwise the contents of the Variable are copied to DAC (2F08).

```
Address... 4EA9H
```

This routine returns the single character pointed to by register pair HL in register A, if it is a lower case alphabetic it converts it to upper case.

```
Address... 4EB8H
```

This routine is used by the Factor Evaluator and the numeric input routine (3299H) to convert an ampersand ("&") Prefixed number from textual form to an integer in DAC. As each legal character is found the product is multiplied by 2, 8 or 16, depending upon the character which initially followed the ampersand, and the new digit added to it. If the product overflows an "`Overflow`" error is generated (4067H). The routine terminates when an unacceptable character is found.

```
Address... 4EFCH
```

This routine is used by the Factor Evaluator to process the FFH prefixed function tokens. If the token is either "`LEFT$`", "`RIGHT$`" or "`MID$`" the string operand is evaluated (4C62H), the address of its descriptor pushed onto the Z80 stack and the following numeric operand also evaluated (521CH) and stacked. Otherwise the function's parenthesized operand is evaluated (4E87H) and, for "`SQR`", "`RND`", "`SIN`", "`LOG`", "`EXP`", "`COS`", "`TAN`" or "`ATN`" only, converted to double precision (303AH). The function token is then used to obtain the required address from the table at 39DEH and control transfers to the function handler.

```
Address... 4F47H
```

This routine is used by the numeric input conversion routine (3299H) to test for a "+" or "-" character or token. It returns register D=0 for positive and register D=FFH for negative.

```
Address... 4F57H
```

This routine is used by the Expression Evaluator to apply a relational operator (<>= or combinations) to a pair of operands. If the operands are numeric the Expression Evaluator first uses the math operator routine (4D22H) to apply the general relation operation to the operands. If the operands are strings the string comparison routine (65C8H) is used first. When control arrives here the relation result is in register A and the Z80 Flags:

```
Operand 1=Operand 2 ... A=00H, Flag Z,NC
Operand 1<Operand 2 ... A=01H, Flag NZ,NC
Operand 1>Operand 2 ... A=FFH, Flag NZ,C
```

The Expression Evaluator also supplies a bit mask defining the original operators on the Z80 stack. This has a 1 in each position if the associated operation is required: 00000<=>. The mask is applied to the relation result producing zero if none of the conditions is satisfied. This is then placed in DAC as a true (-1) or false (0) integer (2E9AH).

```
Address... 4F63H
```

This routine is used by the Factor Evaluator to apply the monadic " `NOT` " operator. Register D is set to an initial precedence level of 5AH and the expression evaluated (4C67H) and converted to an integer (2F8AH). It is then inverted and restored to DAC.

```
Address... 4F78H
```

This routine is used by the Expression Evaluator to apply a logical operator (" `OR` ", " `AND` ", " `XOR` ", " `EQV` " and " `IMP` ") or the " `MOD` " and "" operators to a pair of numeric operands. The first operand, which has already been converted to an integer, is supplied on the Z80 stack and the second is supplied in DAC. The operator token (actually its precedence level) is supplied in register B. After converting the second operand to an integer (2F8AH) the operator is examined. There are separate routines for " `MOD` " (323AH) and "" (31E6H) but the logical operators are processed locally using the corresponding Z80 logical instructions on register pairs DE and HL. The result is stored in DAC as an integer (2F99H).

```
Address... 4FC7H
```

This routine is used by the Factor Evaluator to apply the " `LPOS` " function to an operand contained in DAC. The contents of LPTPOS are placed in DAC as an integer (4FCFH).

```
Address... 4FCCH
```

This routine is used by the Factor Evaluator to apply the " `POS` " function to an operand contained in DAC. The contents of TTYPOS are placed in DAC as an integer (2F99).

```
Address... 4FD5H
```

This routine is used by the Factor Evaluator to apply the " `USR` " function. The user number is collected directly from the program text, it cannot be an expression, and the associated address taken from

USRTAB (4FF4H). The following parenthesized operand is then evaluated (4E87H) and left in DAC as the passed parameter. If it is a string type its storage is freed (67D3H). The current program text position is pushed onto the Z80 stack followed by a return to 3297H, the routine at this address will restore the program text position after the user function has terminated. Control then transfers to the user address with register pair HL pointing to the first byte of DAC and the type code, from VALTYP, in register A. Additionally, for a string parameter, the descriptor address is taken from DAC and placed in register pair DE.

The user routine may modify any register except the Z80 SP and should terminate with a RET instruction, interrupts may be left disabled if necessary as the Runloop will re-enable them. Any numeric parameter to be returned to the Interpreter should be placed in DAC. Strictly speaking this should be the same numeric type as the passed parameter, however if VALTYP is modified the Interpreter will always accept it.

Returning a string type is more difficult. Using the same method as the Factor Evaluator string functions, which involves copying the string to the String Storage Area and pushing a new descriptor onto TEMPST, is complicated and vulnerable to changes in the MSX system. A simpler and more reliable method is to use the passed parameter to create the space for the result. This should not be an explicitly stated string as the program text will have to be modified, instead an implicit parameter should be used. This must be done with care however, it is very easy to gain the impression that the Interpreter has accepted the string when in fact it has not. Take the following example which does nothing but return the passed parameter:

```
10 POKE &H9000,&HC9
20 DEFUSR=&H9000
30 A$=USR(STRING$(12,"!"))
40 PRINT A$
50 B$=STRING$(9,"X")
60 PRINT A$
```

At first it seems that the passed string has been correctly assigned to A$. When line 60 is reached however it becomes apparent that A$ has been corrupted by the subsequent assignment of a string to B$. What has happened is that the temporary storage allocated to the passed parameter was reclaimed from the String Storage Area before control transferred to the user routine. This region was then used to store the string belonging to B$ thus modifying A$.

This situation can be avoided by assigning the parameter to a Variable beforehand and then passing the Variable, for example:

```
10 A$=STRING$(12,"!")
20 A$=USR(A$)
```

Line 10 results in twelve bytes of the String Storage Area being permanently allocated to A$. When the user function is entered the descriptor, which is pointed to by register pair DE, will contain the

starting address of the twelve byte region where the result should be placed. If the returned string is shorter than the passed one the length byte of the descriptor may be changed without any side effects. For further details on string storage see the garbage collector (66B6H).

A point worth noting is that a "`CLEAR`" operation is not strictly necessary before a machine language program is loaded. The region between the top of the Array Storage Area and the base of the Z80 stack is never used by the Interpreter. A program can exist in this region provided that the two enclosing areas do not overlap it.

```
Address... 500EH
```

This is the "`DEFUSR`" statement handler. The user number is collected directly from the program text, it cannot be an expression, and the associated entry in USRTAB located (4FF4H). The address operand is then evaluated (542FH) and placed in USRTAB.

```
Address... 501DH
```

This is the "`DEF FN`" and "`DEFUSR`" statement handler. If the following character is a "`USR`" token (DDH) control transfers to the "`DEFUSR`" statement handler (500EH), otherwise the program text is checked for a trailing "`FN`" token (DEH). The function name Variable is located (51A1H) and, after checking that the Interpreter is in program mode (5193H), the current program text position is placed there. Each of the Variables in the formal parameter list is then located in succession (5EA4H), this is simply to ensure that they are created. The routine terminates by skipping over the remainder of the statement (485BH) as the function body is not required at this time.

```
Address... 5040H
```

This routine is used by the Factor Evaluator to apply the "`FN`" function. The function name Variable is first located (51A1H) to obtain the address of the function definition in the program text. Each formal Variable from the function definition is located in turn (5EA4H) and its address pushed onto the Z80 stack. As each one is found the corresponding actual parameter is evaluated (4C64H) and pushed onto the stack with it. If necessary the type of the actual parameter is converted to match that of the formal parameter (517AH)'

When both lists are exhausted each formal Variable address and actual parameter are popped from the stack in turn. Each Variable is then copied from the Variable Storage Area to PARM2 with its value replaced by the actual parameter. It should be noted that, because PARM2 is only a hundred bytes long, a maximum of nine double precision parameters is allowed. When all the actual parameters have been copied to PARM2 the entire contents of PARM1 (the current parameter area) are pushed onto the Z80 stack and PARM2 is copied to PARM1 (518EH). Register pair HL is then set to the start of the function body in the program text and the expression is evaluated (4C5FH). The old contents of

PARM1 are popped from the stack and restored. Finally the result of the evaluation is type converted if necessary to match the function name type (517AH).

A user defined function differs from a normal expression in only one respect, it has its own set of local Variables. These Variables are created in PARM1 when the function is invoked and disappear when it terminates. When a normal Variable search is initiated by the Expression Evaluator the region examined is the Variable Storage Area. However, if NOFUNS is non-zero, indicating at least one active user function, PARM1 will be searched instead, only if this fails will the search move on to the global Variables in the Variable Storage Area. Using a local Variable area specific to each invocation of a function means that the same Variable names can be used throughout without the Variables overwriting each other or the global Variables.

It is worth noting that a user defined function is slower than an inline expression or even a subroutine. The search carried out to find the function name Variable, plus the large amount of stacking and destacking, are significant overheads.

```
Address... 5189H
```

This routine moves a block of memory from the address pointed to by register pair DE to that pointed to by register pair HL, register pair BC defines the length.

```
Address... 5193H
```

This routine generate an " `Illegal direct` " error if CURLIN shows the Interpreter to be in direct mode.

```
Address... 51A1H
```

This routine checks the program text for an " `FN` " token (DEH) and then creates the function name Variable (5EA9H). These are distinguished from ordinary Variables by having bit 7 set in the first character of the Variable's name.

```
Address... 51ADH
```

Control transfers to this routine from the Runloop execution point (4640H) if a token greater than D8H is found at the start of a statement. If the token is not an FFH prefixed function token a " `Syntax error` " is generated (4055H). If the function token is one of those which double as statements control transfers to the relevant handler, otherwise a " `Syntax error` " is generated. The statements in question are " `MID$` " (696EH), " `STRIG` " (77BFH) and " `INTERVAL` " (77B1H). There is actually no separate token for " `INTERVAL` ", the " `INT` " token (85H) suffices with the remaining characters being checked by the statement handler.

```
Address... 51C9H
```

This is the "`WIDTH`" statement handler. The operand is evaluated (521CH) and its magnitude checked. If it is zero or greater than thirty-two or forty, depending upon the screen mode held in OLDSCR an "`Illegal function call`" error is generated (475AH). If it is the same as the current contents of LINLEN the routine terminates with no further action. Otherwise the current screen is cleared with a FORMFEED control code (0CH) via the OUTDO standard routine in case the screen is to be made smaller. The operand is then placed in LINLEN and either LINL32 or LINL40, depending upon the screen mode held in OLDSCR, and the screen cleared again in case it has been made larger. Because the line length variable to be changed is selected by OLDSCR, rather than SCRMOD, the width can still be changed even if the screen is currently in Graphics Mode or Multicolour Mode. In this case the change is effective when a return is made to the Interpreter Mainloop or an "`INPUT`" statement is executed.

```
Address... 520EH
```

This routine evaluates the next expression in the program text (4C64H), converts it to an integer (2F8AH) and places the result in register pair DE. The magnitude and sign of the MSB are then tested and the routine terminates.

```
Address... 521BH
```

This routine evaluates the next operand in the program text (4C64H) and converts it to an integer (5212H). If the operand is greater than 255 an "`Illegal function call`" error is generated (475AH).

```
Address... 5229H
```

This is the "`LLIST`" statement handler. PRTFLG is set to 01H, to direct output to the printer, and control drops into the "`LIST`" statement handler.

```
Address... 522EH
```

This is the "`LIST`" statement handler. The optional start and termination line number operands are collected and the starting position found in the program text (4279H). Successive program lines are listed until the end link is found, the CTRL-STOP key is pressed or the termination line number is reached, control then transfers directly to the Mainloop "`OK`" point (411FH). Each program line is listed by displaying its line number (3412H), detokenizing (5284H) and displaying (527BH) the line itself and then issuing a CR,LF (7328H).

```
Address... 5284H
```

This routine is used by the "`LIST`" statement handler to convert a tokenized program line to textual form. On entry register pair HL points to the first character of the tokenized line. On exit the line of text is in BUF and is terminated by a zero byte.

Any normal or FFH prefixed token is converted to the corresponding keyword by a simple linear search of the tokens in the table at 3A72H. Exceptions are made if either an opening quote character, a "`REM`" token, or a "`DATA`" token has previously been found. Normally these tokens will be followed by plain text anyway, the check is made to stop graphics characters being interpreted as tokens. The three byte sequence ":" (3AH), "`REM`" token (8FH), " " token (E6H) is converted to the single " " character (27H) and the statement separator (3AH) preceding an "`ELSE`" token (A1H) is scrubbed out.

If one of the numeric tokens is found its value and type are first copied from CONLO and CONTYP to DAC and VALTYP (46E8H). It is then converted to textual form in FBUFFR by the decimal (3425H), octal (371EH) or hex (3722H) conversion routines. For octal and hex types the number is prefixed by an ampersand and an "`O`" or "`H`" letter. A type suffix, "!" or "#", is added to single precision or double precision numbers only if there is no decimal part and no exponent part ("`E`" or "`D`").

```
Address... 53E2H
```

This is the "`DELETE`" statement handler. The optional start and termination line number operands are collected and the starting position found in the program text (4279H). If any pointers exist in the program text they are converted back to line numbers (54EAH). The terminating program line is found by a search of the program text (4295H), if this address is smaller than that of the starting program line an "`Illegal function call`" error is generated (475AH), otherwise the message "`OK`" is displayed (6678H). The block of memory from the end of the terminating line to the start of the Variable Storage Area is copied down to the beginning of the starting line and VARTAB, ARYTAB and STREND are reset to the new (lower) end of the program text. Control then transfers directly to the end of the Mainloop (4237H) to reset the remaining pointers and to relink the Program Text Area. Note that, because control does not return to the normal "`OK`" point, the screen will not be returned to text mode. If the screen is in Graphics Mode or Multicolour mode when a "`DELETE`" is executed, which is admittedly rather unlikely, the system will crash.

```
Address... 541CH
```

This routine is used by the Factor Evaluator to apply the "`PEEK`" function to an operand contained in DAC. The address operand is checked (5439H) then the byte read from memory and placed in DAC as an integer (4FCFH).

```
Address... 5423H
```

This is the "`POKE`" statement handler. The address operand is evaluated (542FH) then the data operand evaluated (521CH) and written to memory.

```
Address... 542FH
```

This routine evaluates the next operand in the program text (4C64H) and places it in register pair DE as an integer (5439H).

```
Address... 5439H
```

This routine converts the numeric operand contained in DAC into an integer in register pair HL. The operand must be in the range -32768 to +65535 and is normally an address as required by "`POKE`", "`PEEK`", "`BLOAD`", etc. The operand's type is first checked via the GETYPR standard routine, if it is already an integer it is simply placed in register pair HL (2F8AH). Assuming the operand is single precision or double precision its sign is checked, if it is negative it is converted to integer (2F8AH). Otherwise it is converted to single precision (2FB2H) and its magnitude checked (2F21H). If it is greater than 32767 and smaller than 65536 then -65536 is added (324EH) before it is converted to integer (2F8AH).

```
Address... 5468H
```

This is the "`RENUM`" statement handler. If a new initial line number operand exists it is collected (475FH), otherwise a default value of ten of taken. If an old initial line number operand exists it is collected (475FH), otherwise a default value of zero is taken. If an increment line number operand exists it is collected (4769H), otherwise a default value of ten is taken.

The program text is then searched for existing line numbers equal to or greater than the new initial line number (4295H) and the old initial line number (4295H), an "`Illegal function call`" error is generated (475AH) if the new address is smaller than the old address. This is to catch any attempt to renumber high program lines down to existing low ones.

A dummy renumbering run of the program text is first carried out to check than no new line number will be generated with a value greater than 65529. This must be done as an error midway through the conversion would leave the program text in a confused state. Assuming all is well any line number operands in the program text are converted to pointers (54F6H). This neatly solves the problem of line number references, `GOTO 50` for example, as the program text is not moved during renumbering. Starting at the old initial program text position each existing program line number is replaced with its new value. When the end link is reached any program text pointers are converted back to line number operands (54F1H) and control transfers directly to the Mainloop "`OK`" point (411EH).

```
Address... 54F6H
```

When entered at 54F6H this routine converts every line number operand in the program text to a pointer. When entered at 54F7H it performs the reverse operation and converts every pointer in the program text back to a line number operand. Starting at the beginning of the Program Text Area each line is examined for a pointer token (0DH) or a line number operand token (0EH) depending upon the mode. In pointer to line number operand mode the pointer is replaced by the line number from the referenced program line and the token changed to 0EH. In line number operand to pointer mode the program text is searched (4295H) to find the relevant line, its address replaces the line number operand and the token is changed to 0DH. If the search is unsuccessful a message of the form "`Undefined line NNNN in NNNN`" is displayed (6678H) and the conversion process continues. A special check is made for the "`ON ERROR GOTO 0`" statement, to prevent the generation of a spurious error message, but no check is made for the similar "`RESUME 0`" statement. In this case an error message will be displayed, this should be ignored.

```
Address... 555AH
```

This is the plain text message "`Undefined line`" terminated by a zero byte.

```
Address... 558CH
Name...... SYNCHR
Entry..... HL points to character to check
Exit...... A=Next program character
Modifies.. AF, HL
```

Standard routine to check the current program text character, whose address is supplied in register pair HL, against a reference character. The reference character is supplied as a single byte immediately following the `CALL` or `RST` instruction, for example:

```
RST 08H
DEFB ","
```

If the characters do not match a "`Syntax error`" is generated (4055H), otherwise control transfers to the CHRGTR standard routine to fetch the next program character (4666H).

```
Address... 5597H
Name...... GETYPR
Entry..... None
Exit...... AF=Type
Modifies.. AF
```

Standard routine to return the type of the current operand, determined by VALTYP, as follows:

```
Integer..............A=FFH, Flag M,NZ,C
String...............A=00H, Flag P,Z,C
```

```
Single Precision ... A=01H, Flag P,NZ,C
Double Precision ... A=05H, Flag P,NZ,NC
```

```
Address... 55A8H
```

This is the "`CALL`" statement handler. The extended statement name, which is an unquoted string up to fifteen characters long terminated by a "(", ":" or end of line character (00H), is first copied from the program text to PROCNM, any unused bytes are zero filled. Bit 5 of each entry in SLTATR is then examined for an extension ROM with a statement handler. If a suitable ROM is found its position in SLTATR is converted to a Slot ID in register A and a ROM base address in register H (7E2AH). The statement handler address is read from ROM locations four and five (7E1AH) and placed in register pair IX. The Slot ID is placed in the high byte of register pair IY and the ROM statement handler called via the CALSLT standard routine.

The ROM will examine the statement name and return Flag C if it does not recognize it, otherwise it performs the required operation. If the ROM call fails the search of SLTATR continues until the table is exhausted whereupon a "`Syntax error`" is generated (4055H). If the ROM call is successful the handler terminates.

```
Address... 55F8H
```

This routine is used by the device name parser (6F15H) when it cannot recognize a device name found in the program text. Upon entry register pair HL points to the first character of the name and register B holds its length. The name is first copied to PROCNM and terminated by a zero byte. Bit 6 of each entry in SLTATR is then examined for an extension ROM with a device handler. If a suitable ROM is found its position in SLTATR is converted to a Slot ID in register A and a ROM base address in register H (7E2AH). The device handler address is read from ROM locations six and seven (7E1AH) and placed in register pair IX. The Slot ID is placed in the high byte of register pair IY, the unknown device code (FFH) in register A and the ROM device handler called via the CALSLT standard routine.

The ROM will examine the device name and return Flag C if it does not recognize it, otherwise it returns its own internal code from zero to three. If the ROM call fails the search of SLTATR continues until the table is exhausted whereupon a "`Bad file name`" error is generated (6E6BH). If the ROM call is successful the ROM's internal code is added to its SLTATR position, multiplied by a factor of four, to produce a global device code' The base code for each entry in SLTATR is shown below in hexadecimal. The "SS" and "PS" markers show the corresponding Secondary and Primary Slot numbers, each slot is composed of four pages:

SS0				SS1				SS2				SS3				
00	04	08	0C	10	14	18	1C	20	24	28	2C	30	34	38	3C	PS0
40	44	48	4C	50	54	58	5C	60	64	68	6C	70	74	78	7C	PS1
80	84	88	8C	90	94	98	9C	A0	A4	A8	AC	B0	B4	B8	BC	PS2
C0	C4	C8	CC	D0	D4	D8	DC	E0	E4	E8	EC	F0	F4	F8	FC	PS3

Figure 44: Device Codes

The global device code is used by the Interpreter until the time comes for the ROM to perform an actual device operation. It is then converted back into the ROM's Slot ID, base address and internal device code to perform the ROM access. Note that the codes from 0 to 8 are reserved for disk drive identifiers and those from FCH to FFH for the standard devices GRP, CRT, LPT and CAS. With the current MSX hardware structure these codes correspond to physically improbable ROM configurations and are therefore safe to be used for specific purposes by the Interpreter.

```
Address... 564AH
```

This routine is used by the function dispatcher (6F8FH) when it encounters a device code not belonging to one of the standard devices. The device code is first converted to a SLTATR position and then to a Slot ID in register A and ROM base address in register H (7E2DH). The ROM device handler address is read from ROM locations six and seven (7E1AH) and placed in register pair IX. The Slot ID is placed in the high byte of register pair IY, the ROM's internal device code in DEVICE and the ROM device handler called via the CALSLT standard routine.

```
Address... 566CH
```

This entry point to the macro language parser is used by the "`DRAW`" statement handler, a later entry point (56A2H) is used by the "`PLAY`" statement handler. The command string is evaluated (4C64H) and its storage freed (67D0H). After pushing a zero termination block onto the Z80 stack the length and address of the string body are placed in MCLLEN and MCLPTR and control drops into the parser mainloop.

```
Address... 56A2H
```

This is the macro language parser mainloop, it is used to process the command string associated with a "`DRAW`" or "`PLAY`" statement' On entry the string length is in MCLLEN, the string address is in MCLPTR and the address of the relevant command table is in MCLTAB. The command tables contain the legal command letters, together with the associated command handler addresses, for each statement. The "`DRAW`" table is at 5D83H and the "`PLAY`" table at 752EH.

The parser mainloop first fetches the next character from the command string (56EEH). If there are no more characters left the next string descriptor is popped from the stack (568CH). If this is zero the parser terminates (5709H) if MCLFLG shows a "`DRAW`" statement to be active, otherwise control transfers back to the "`PLAY`" statement handler (7494H).

Assuming a command character exists the current command table is searched to check its legality, if no match is found an "`Illegal function call`" error is generated (475AH). The command table entry is then examined, if bit 7 is set the command takes an optional numeric parameter. If this is present it is collected and placed in register pair DE (571CH), otherwise a default value of one is taken. After pushing a return to the start of the parser mainloop onto the Z80 stack control transfers to the command handler at the address taken from the command table.

```
Address... 56EEH
```

This routine is used by the macro language parser to fetch the next character from the command string. If MCLLEN is zero the routine terminates with Flag Z, there are no characters left. Otherwise the next character is taken from the address contained in MCLPTR and returned in register A, if the character is lower case it is converted to upper case. MCLPTR is then incremented and MCLLEN decrement Ed.

```
Address... 570BH
```

This routine is used by the macro language parser to return an unwanted character to the command string. MCLLEN is incremented and MCLPTR decremented.

```
Address... 5719H
```

This routine is used by the macro language parser to collect a numeric parameter from the command string. The result is a signed integer and is returned in register pair DE, it cannot be an expression. The first character is taken and examined, if it is a "+" it is ignored and the next character taken (5719H). If it is a "-" a return is set up to the negation routine (5795H) and the next character taken (5719H). If it is an "=" the value of the following Variable is returned (577AH). Otherwise successive characters are taken and a binary product accumulated until a non-numeric character is found.

```
Address... 575AH
```

This routine is used by the macro language parser "=" and "X" handlers. The Variable name is copied to BUF until the ";" delimiter is found, if this takes more than thirty-nine characters to find an "`Illegal function call`" error is generated (475AH). Otherwise control transfers to the Factor Evaluator Variable handler (4E9BH) and the Variable contents are returned in DAC.

```
Address... 577AH
```

This routine is used by the macro language parser to process the "=" character in a command parameter. The Variable's value is obtained (575AH), converted to an integer (2F8AH) and placed in register pair DE.

```
Address... 5782H
```

This routine is used by the macro language parser to process the "X" command. The Variable is processed (575AH) and, after checking that stack space is available (625EH), the current contents of MCLLEN and MCLPTR are stacked. Control then transfers to the parser entry point (5679H) to obtain the Variable's descriptor and process the new command string.

```
Address... 579CH
```

This routine is used by various graphics statements to evaluate a coordinate pair in the program text. The coordinates must be parenthesized with a comma separating the component operands. If the coordinate pair is preceded by a " `STEP` " token (DCH) each component value is added to the corresponding component of the current graphics coordinates in GRPACX and GRPACY, otherwise the absolute values are returned. The X coordinate is returned in GRPACX, GXPOS and register pair BC. The Y coordinate is returned in GRPACY, GYPOS and register pair DE.

There are two entry points to the routine, the one which is used depends on whether the caller is expecting more than one coordinate pair. The " `LINE` " statement, for example, expects two coordinate pairs the first of which is the more flexible. The entry point at 579CH is used to collect the first coordinate pair and will accept the characters "-" or "@-" as representing the current graphics coordinates. The entry point at 57ABH is used for the second coordinate pair and requires an explicit operand.

```
Address... 57E5H
```

This is the " `PRESET` " statement handler. The current background colour is taken from BAKCLR and control drops into the " `PSET` " handler.

```
Address... 57EAH
```

This is the " `PSET` " statement handler. After the coordinate pair has been evaluated (57ABH) the current foreground colour is taken from FORCLR and used as the default when setting the ink colour (5850H). The current graphics coordinates are converted to a physical address, via the SCALXY and MAPXYC standard routines, and the colour of the current pixel set via the SETC standard routine.

```
Address... 5803H
```

This routine is used by the Factor Evaluator to apply the " `POINT` " function. The current contents of CLOC, CMASK, GYPOS, GXPOS, GRPACY and GRPACX are stacked and the coordinate pair operand evaluated (57ABH). The colour of the new pixel is read via the SCALXY, MAPXYC and READC standard routines and placed in DAC as an integer (2F99H), the old coordinate values are then popped and restored. Note that a value of -1 is returned if the point coordinates are outside the screen.

```
Address... 5850H
```

This graphics routine is used to evaluate an optional colour operand in the program text and to make it the current ink colour. After checking the screen mode (59BCH) the colour operand is evaluated (521CH) and placed in ATRBYT. If no operand exists the colour code supplied in register A is placed in ATRBYT instead.

```
Address... 5871H
```

This graphics routine returns the difference between the contents of GXPOS and register pair BC in register pair HL. If the result is negative (GXPOS<BC) it is negated to produce the absolute magnitude and Flag C is returned.

```
Address... 5883H
```

This graphics routine returns the difference between the contents of GYPOS and register pair DE in register pair HL. If the result is negative (GYPOS<DE) it is negated to produce the absolute magnitude and Flag C is returned.

```
Address... 588EH
```

This graphics routine swaps the contents of GYPOS and register pair DE.

```
Address... 5898H
```

This graphics routine first swaps the contents of GYPOS and register pair DE (588EH) then swaps the contents of GXPOS and register pair BC. When entered at 589BH only the second operation is performed.

```
Address... 58A7H
```

This is the "`LINE`" statement handler. The first coordinate pair (X1,Y1) is evaluated (579CH) and placed in register pairs BC,DE. After checking for the "-" token (F2H) the second coordinate pair (X2,Y2) is evaluated (57ABH) and left in GRPACX, GRPACY and GXPOS, GYPOS. After setting the ink colour (584DH) the program text is checked for a following "B" or "BF" option and either the box (5912H), boxfill (58BFH) or linedraw (58FCH) operation performed. None of these operations affects the current graphics coordinates in GRPACX and GRPACY, these are left at X2,Y2.

```
Address... 58BFH
```

This routine performs the boxfill operation. Given that the supplied coordinate pairs define diagonally opposed points of the box two quantities must be derived from them. The horizontal size of the box is obtained from the difference between X1 and X2, this gives the number of pixels to set per row. The vertical size is obtained from the difference between Y1 and Y2 giving the number of rows required. Starting at the physical address of X1,Y1, and moving successively lower via the DOWNC standard routine, the required number of pixel rows are filled in by repeated use of the NSETCX standard routine.

```
Address... 58FCH
```

This routine performs the linedraw operation. After drawing the line (593CH) GXPOS and GYPOS are reset to X2,Y2 from GRPACX and GRPACY.

```
Address... 5912H
```

This routine performs the box operation. The box is produced by drawing a line (58FCH) between each of the four corner points. The coordinates of each corner are derived from the initial operands by interchanging the relevant component of the pair. The drawing sequence is:

1. X1,Y2 to X2,Y2
2. X1,Y1 to X2,Y1
3. X2,Y1 to X2,Y2
4. X1,Y1 to X1,Y2

```
Address... 593CH
```

This routine draws a line between the points X1,Y1, supplied in register pairs BC and DE and X2,Y2, supplied in GXPOS and GYPOS. The operation of the drawing mainloop (5993H) is best illustrated by an example, say LINE(0,0)-(10,4). To reach the end point of the line from its start ten horizontal steps (X2- X1) and four downward steps (Y2-Y1) must be taken altogether. The best approximation to a straight line therefore requires two and a half horizontal steps for every downward step (X2-X1/Y2-

Y1). While this is impossible in practice, as only integral steps can be taken, the correct ratio can be achieved on average.

The method employed is to add the Y difference to a counter each time a rightward step is taken. When the counter exceeds the value of the X difference it is reset and one downward step is taken, this is in effect an integer division of the two difference values. Sometimes downward steps will be produced every two rightward steps and sometimes every three rightward steps. The average, however, will be one downward step every two and a half rightward steps. An equivalent BASIC program is shown below with a slightly offset BASIC line for comparison:

```
10 SCREEN 0
20 INPUT"START X,Y";X1,Y1
30 INPUT"END X,Y",X2,Y2
40 SCREEN 2
50 X=X1:Y=Y1:L=X2-X1:S=Y2-Y1:CTR=L/2
60 PSET(X,Y)
70 CTR=CTR+S:IF CTR<L THEN 90
80 CTR=CTR-L:Y=Y+1
90 X=X+1:IF X<=X2 THEN 60
100 LINE(X1,Y1+5)-(X2,Y2+5)
110 GOTO 110
```

The above example suffers from three limitations. The line must slope downwards, it must slope to the right and the slope cannot exceed forty-five degrees from the horizontal (one downward step for one rightward step).

The routine overcomes the first limitation by examining the Y1 and Y2 coordinates before drawing commences. If Y2 is greater than or equal to Y1, showing the line to slope upwards or to be horizontal, both coordinate pairs are exchanged. The line is now sloping downwards and will be drawn from the end point to the start.

The second limitation is overcome by examining X1 and X2 beforehand to determine which way the line is sloping. If X2 is greater than or equal to X1 the line slopes to the right and a Z80 JP to the RIGHTC standard routine is placed in MINUPD/MAXUPD (see below) for use by the drawing mainloop, otherwise a JP to the LEFTC standard routine is placed there.

The third limitation is overcome by comparing the X coordinate difference to the Y coordinate difference before drawing to determine the slope steepness. If X2-X1 is smaller than Y2-Y1 the slope of the line is less than forty-five degrees from the horizontal. The simple method shown above for LINE(0,0)-(10,4) will not work for slopes greater than forty- five degrees as the maximum rate of descent is achieved when one downward step is taken for every horizontal step. It will work however if the step directions are exchanged. Thus LINE(0,0)-(4,10) requires one rightward step for every two and a half downward steps. MINUPD holds a Z80 JP to the "normal" step direction standard routine for the drawing mainloop and MAXUPD holds a JP to the "slope" step direction standard routine. For shallow angles MINUPD will vector to DOWNC and MAXUPD to LEFTC or RIGHTC. For steep angles MINUPD will vector to LEFTC or RIGHTC and MAXUPD to DOWNC. For steep angles the counter

values must also be exchanged, the X difference must now be added to the counter and the Y difference used as the counter limit. The variables MINDEL and MAXDEL are used by the drawing mainloop to hold these counter values, MINDEL holds the smaller end point difference and MAXDEL the larger.

An interesting point is that the reference counter, held in CTR in the above program and in register pair DE in the ROM, is preloaded with half the largest end point difference rather than being set to zero. This has the effect of splitting the first "stair" in the line into two sections, one at the start of the line and one at its end, and improving the line's appearance.

```
Address... 59B4H
```

This graphics routine shifts the contents of register pair DE one bit to the right.

```
Address... 59BCH
```

This routine generates an "`Illegal function call`" error (475AH) if the screen is not in Graphics Mode or Multicolour Mode.

```
Address... 59C5H
```

This is the "`PAINT`" statement handler. The starting coordinate pair is evaluated (579CH), the ink colour set (584DH) and the optional boundary colour operand evaluated (521CH) and placed in BDRATR. The starting coordinate pair is checked to ensure that it is within the screen (5E91H) and is made the current pixel physical address by the MAPXYC standard routine. The distance to the right hand boundary is then measured (5ADCH) and, if it is zero, the handler terminates. Otherwise the distance to the left hand boundary is measured (5AEDH) and the sum of the two placed in register pair DE as the zone width. The current position is then stacked twice (5ACEH), first with a termination flag (00H) and then with a down direction flag (40H). Control then transfers to the paint mainloop (5A26H) with an up direction flag (C0H) in register B.

```
Address... 5A26H
```

This is the paint mainloop. The zone width is held in register pair DE, the paint direction, up or down, in register B and the current pixel physical address is that of the pixel adjacent to the left hand boundary. A vertical step is taken to the next line, via the TUPC or TDOWNC standard routines, and the distance to the right hand boundary measured (5ADCH). The distance to the left hand boundary is then measured and the line between the boundaries filled in (5AEDH). If no change is found in the position of either boundary control transfers to the start of the mainloop to continue painting in the same direction. If a change is found an inflection has occurred and the appropriate action must be taken.

There are four types of inflection, LH or RH incursive, where the relevant boundary moves inward, and LH or RH excursive, where it moves outward. An example of each type is shown below with numbered zones indicating the order of painting during upward movement. A secondary zone is shown within each inflective region for completeness:

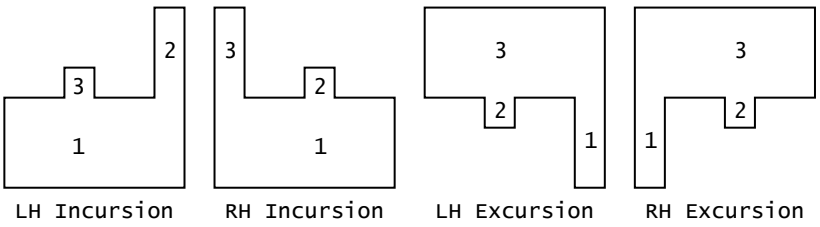


Figure 45: Boundary Inflections

A LH excursion has occurred when the distance to the left hand boundary is non-zero, a RH excursion has occurred when the current zone width is greater than that of the previous line. Unless the excursion is less than two pixels, in which case it will be ignored, the current position (the bottom left of zone 3 in figure 45) is stacked (5AC2H), the paint direction reversed and painting restarts at the top left of the excursive region .

A RH incursion has occurred when the current zone width is smaller than that of the previous line. If the incursion is total, that is the current zone width is zero, a dead end has been reached and the last position and direction are popped (5AIFH) and painting restarts at that point. Otherwise the current position and direction are stacked (5AC2H) and painting restarts at the bottom left of the incursive region.

A LH incursion is dealt with automatically during the search for the right hand boundary and requires no explicit action by the paint mainloop.

```
Address... 5AC2H
```

This routine is used by the "`PAINT`" statement handler to save the current paint position and direction on the Z80 stack. The six byte parameter block is made up of the following:

```
2 bytes ... Current contents of CLOC
1 byte  ... Current direction
1 byte  ... Current contents of CMASK
2 bytes ... Current zone width
```

After the parameters have been stacked a check is made that sufficient stack space still exists (625EH).

```
Address... 5ADCH
```

This routine is used by the "`PAINT`" statement handler to locate the right hand boundary. The zone width of the previous line is passed to the SCANR standard routine in register pair DE, this determines the maximum number of boundary colour pixels that may initially be skipped over. The returned skip count remainder is placed in SKPCNT and the number of non-boundary colour pixels traversed in MOVCNT.

```
Address... 5AEDH
```

This routine is used by the "`PAINT`" statement handler to locate the left hand boundary. The end point of the right hand boundary search is temporarily saved and the starting point taken from CSAVEA and CSAVEM and made the current pixel physical address. The left hand boundary is then located via the SCANL standard routine, which also fills in the entire zone, and the right hand end point recovered and placed in CSAVEA and CSAVEM.

```
Address... 5B0BH
```

This routine is used by the "`CIRCLE`" statement handler to negate the contents of register pair DE.

```
Address... 5B11H
```

This is the "`CIRCLE`" statement handler. After evaluating the centre coordinate pair (579CH) the radius is evaluated (520FH), multiplied (325CH) by SIN(PI/4) and placed in CNPNTS. The ink colour is set (584DH), the start angle evaluated (5D17H) and placed in CSTCNT and the end angle evaluated (5D17H) and placed in CENCNT. If the end angle is smaller than the start angle the two values are swapped and CPLOTF is made non-zero. The aspect ratio is evaluated (4C64H) and, if it is greater than one, its reciprocal is taken (3267H) and CSCLXY is made non-zero to indicate an X axis squash. The aspect ratio is multiplied (325CH) by 256, converted to an integer (2F8AH) and placed in ASPECT as a single byte binary fraction. Register pairs HL and DE are set to the starting position on the circle perimeter (X=RADIUS,Y=0) and control drops into the circle mainloop.

```
Address... 5BBDH
```

This is the circle mainloop. Because of the high degree of symmetry in a circle it is only necessary to compute the coordinates of the arc from zero to forty-five degrees. The other seven segments are produced by rotation and reflection of these points. The parametric equation for a unit circle, with T the angle from zero to PI/4, is:

```
X=COS(T)
Y=SIN(T)
```

Direct computation using this equation, or the corresponding functional form X=SQR(1-Y^2), is too slow, instead the first derivative is used:

```
 dx
---- = -Y/X
 dy
```

Given that the starting position is known (X=RADIUS,Y=0), the X coordinate change for each unit Y coordinate change may be computed using the derivative. Furthermore, because graphics resolution is limited to one pixel, it is only necessary to know when the sum of the X coordinate changes reaches unity and then to decrement the X coordinate. Therefore:

```
Decrement X when (Y1/X)+(Y2/X)+(Y3/X)+... => 1
Therefore decrement when (Y1+Y2+Y3+...)/X => 1
Therefore decrement when     Y1+Y2+Y3+... => X
```

All that is required to identify an X coordinate change is to totalize the Y coordinate values from each step until the X coordinate value is exceeded. The circle mainloop holds the X coordinate in register pair HL, the Y coordinate in register pair DE and the running total in CRCSUM. An equivalent BASIC program for a circle of arbitrary radius 160 pixels is:

```
10 SCREEN 2
20 X=160:Y=0:CRCSUM=0
30 PSET(X,191-Y)
40 CRCSUM=CRCSUM+Y :Y=Y+1
50 IF CRCSUM<X THEN 30
60 CRCSUM=CRCSUM-X:X=X-1
70 IF X>Y THEN 30
80 CIRCLE(0,191),155
90 GOTO 90
```

The coordinate pairs generated by the mainloop are those of a "virtual" circle, such tasks as axial reflection, elliptic squash and centre translation are handled at a lower level (5C06H).

```
Address... 5C06H
```

This routine is used to by the circle mainloop to convert a coordinate pair, in register pairs HL and DE, into eight symmetric points on the screen. The Y coordinate is initially negated (5B0BH), reflecting it about the X axis, and the first four points produced by successive clockwise rotations through ninety degrees (5C48H). The Y coordinate is then negated again (5B0BH) and a further four points produced (5C48H).

Clockwise rotation is performed by exchanging the X and Y coordinates and negating the new Y coordinate, thus a point (40,10) would become (10,-40). Assuming an aspect ratio of 0.5, for example,

the complete sequence of eight points would therefore be:

1. X,-Y*0.5
2. -Y,-X*0.5
3. -X, Y*0.5
4. Y, X*0.5
5. Y,-X*0.5
6. -X,-Y*0.5
7. -Y, X*0.5
8. X, Y*0.5

It can be seen from the above that, ignoring the sign of the coordinates for the moment, there are only four terms involved. Therefore, rather than performing the relatively slow aspect ratio multiplication (5CEBH) for each point, the terms X*0.5 and Y*0.5 can be prepared in advance and the complete sequence generated by interchanging and negating the four terms. With the aspect ratio shown above the initial conditions are set up so that register pair HL=X, register pair DE=-Y*0.5, CXOFF=Y and CYOFF=X*0.5 and successive points are produced by the operations:

1. Exchange HL and CXOFF, negate HL.
2. Exchange DE and CYOFF, negate DE.

In parallel with the computation of each circle coordinate the number of points required to reach the start of the segment containing the point is kept in CPCNT8. This will initially be zero and will increase by 2*RADIUS*SIN(PI/4) as each ninety degree rotation is made. As each of the eight points is produced its Y coordinate value is added to the contents of CPCNT8 and compared to the start and end angles to determine the appropriate course of action. If the point is between the two angles and CPLOTF is zero, or if it is outside the angles and CPLOTF is non-zero, the coordinates are added to the circle centre coordinates (5CDCH) and the point set via the SCALXY, MAPXYC and SETC standard routines. If the point is equal to either of the two angles, and the associated bit is set in CLINEF, the coordinates are added to the circle centre coordinates (5CDCH) and a line drawn to the centre (593CH). If none of these conditions is applicable no action is taken other than to proceed to the next point.

```
Address... 5CEBH
```

This routine multiplies the coordinate value supplied in register pair DE by the aspect ratio contained in ASPECT, the result is returned in register pair DE. The standard binary shift and add method is used but the operation is performed as two single byte multiplications to avoid overflow problems.

```
Address... 5D17H
```

This routine is used by the "`CIRCLE`" statement handler to convert an angle operand to the form required by the circle mainloop, the result is returned in register pair DE. While the method used is basically sound, and eliminates one trigonometric computation per angle, the results produced are inaccurate. This is demonstrated by the following example which draws a line to the true thirty degree point on a circle's perimeter:

```
10 SCREEN 2
20 PI = 4 * ATN(1)
30 CIRCLE(100,100),80,,PI/6
40 LINE(100,100)-(100+80*COS(PI/6),100-80*SIN(PI/6))
50 GOTO 50
```

The result that the routine should produce is the number of points that must be produced by the circle mainloop before the required angle is reached. This can be computed by first noting that there will be INT(ANGLE/(PI/4)) forty-five degree segments prior to the segment containing the required angle. Furthermore each forty-five segment will contain RADIUS*SIN(PI/4) points as this is the value of the terminating Y coordinate. Therefore the number of points required to reach the start of the segment containing the angle is the product of these two numbers. The total count is produced by adding this figure to the number of points required to cover any remaining angle within the final segment, that is RADIUS*SIN(REMAINING ANGLE) points.

Unfortunately the routine computes the number of points within a segment by linear approximation from the total segment size on the mistaken assumption that successive points subtend equal angles. Thus in the above example the point count computed for the angle is 30/45*(80*0.707107)=37 instead of the correct value of forty. The error produced by the routine is therefore at a maximum at the centre of each forty-five degree segment and reduces to zero at the end points.

```
Address... 5D6EH
```

This is the "`DRAW`" statement handler. Register pair DE is set to point to the command table at 5D83H and control transfers to the macro language parser (566CH).

```
Address... 5D83H
```

This table contains the valid command letters and associated addresses for the "`DRAW`" statement commands. Those commands which takes a parameter, and consequently have bit 7 set in the table, are shown with an asterisk:

CMD	TO
U*	5DB1H
D*	5DB4H

CMD	TO
L*	5DB9H
R*	5DBCH
M	5DD8H
E*	5DCAH
F*	5DC6H
G*	5DD1H
H*	5DC3H
A*	5E4EH
B	5E46H
N	5E42H
X	5782H
C*	5E87H
S*	5E59H

```
Address... 5DB1H
```

This is the "`DRAW`" statement "`U`" command handler. The operation of the "`D`", "`L`", "`R`", "`E`", "`F`", "`G`" and "`H`" commands is very similar so no separate description of their handlers is given. The optional numeric parameter is supplied by the macro language parser in register pair DE. This initial parameter is modified by a given handler into a horizontal offset in register pair BC and a vertical offset in register pair DE. For example if leftward or upward movement is required the parameter is negated (5B0BH), if diagonal movement is required the parameter is duplicated so that equal horizontal and vertical offsets are produced. Once the offsets have been prepared control transfers to the line drawing routine (5DFFH).

```
Address... 5DD8H
```

This is the "`DRAW`" statement "`M`" command handler. The character following the command letter is examined then the two parameters collected from the command string (5719H). If the initial character is "+" or "-" the parameters are regarded as offsets and are scaled (5E66H), rotated through successive ninety degree steps as determined by DRWANG and then added to the current graphics coordinates (5CDCH) to determine the termination point. If DRWFLG shows the "`B`" mode to be inactive a line is then drawn (5CCDH) from the current graphics coordinates to the termination point.

If DRWFLG shows the " `N` " mode to be inactive the termination coordinates are placed in GRPACX and GRPACY to become the new current graphics coordinates. Finally DRWFLG is zeroed, turning the " `B` " and " `N` " modes off, and the handler terminates.

```
Address... 5E42H
```

This is the " `DRAW` " statement " `N` " command handler, DRWFLG is simply set to 40H.

```
Address... 5E46H
```

This is the " `DRAW` " statement " `B` " command handler, DRWFLG is simply set to 80H.

```
Address... 5E4EH
```

This is the " `DRAW` " statement " `A` " command handler. The parameter is checked for magnitude and placed in DRWANG.

```
Address... 5E59H
```

This is the " `DRAW` " statement " `S` " command handler. The parameter is checked for magnitude and placed in DRWSCL.

```
Address... 5E66H
```

This routine is used by the " `DRAW` " statement " `U` ", " `D` ", " `L` ", " `R` ", " `E` ", " `F` ", " `G` ", " `H` " and " `M` " (in offset mode) command handlers to scale the offset supplied in register pair DE by the contents of DRWSCL. Unless DRWSCL is zero, in which case the routine simply terminates, the offset is multiplied using repeated addition and then divided by four (59B4H). To eliminate scaling an " `S0` " or " `S4` " command should be used.

```
Address... 5E87H
```

This is the " `DRAW` " statement " `C` " command handler. The parameter is placed in ATRBYT via the SETATR standard routine. There is no check on the MSB of the parameter so illegal values such as " `C265` " will be accepted without an error message.

```
Address... 5E91H
```

This routine is used by the " `PAINT` " statement handler to check, via the SCALXY standard routine, that the coordinates in register pairs BC and DE are within the screen. If not an " `Illegal function`

`call` " error is generated (475AH).

```
Address... 5E9FH
```

This is the " `DIM` " statement handler. A return is set up to 5E9AH, so that multiple Arrays can be processed, DIMFLG is made non-zero and control drops into the Variable search routine.

```
Address... 5EA4H
```

This is the Variable search routine. On entry register pair HL points to the first character of the Variable name in the program text. On exit register pair HL points to the character following the name and register pair DE to the first byte of the Variable contents in the Variable Storage Area. The first character of the name is taken from the program text, checked to ensure that it is upper case alphabetic (64A7H) and placed in register C. The optional second character, with a default value of zero, is placed in register B, this character may be alphabetic or numeric. Any further alphanumeric characters are then simply skipped over. If a type suffix character ("%", "$", "!" or "#") follows the name this is converted to the corresponding type code (2, 3, 4 or 8) and placed in VALTYP. Otherwise the Variable's default type is taken from DEFTBL using the first letter of the name to locate the appropriate entry.

SUBFLG is then checked to determine how any parenthesized subscript following the name should be treated. This flag is normally zero but is modified by the " `ERASE` " (01H), " `FOR` " (64H), " `FN` " (80H) or " `DEF FN` " (80H) statement handlers to force a particular course of action. In the " `ERASE` " case control transfers straight to the Array search routine (5FE8H), no parenthesized subscript need be present. In the " `FOR` ", " `FN` " and " `DEF FN` " cases control transfers straight to the simple Variable search routine (5F08H), no check is made for a parenthesized subscript. Assuming that the situation is normal the program text is checked for the characters "(" or "[". If either is present control transfers to the Array search routine (5FBAH), otherwise control drops into the simple Variable search routine.

```
Address... 5F08H
```

This is the simple Variable search routine. There are four types of simple Variable each composed of a header followed by the Variable contents. The first byte of the header contains the type code and the next two bytes the Variable name. The contents of the Variable will be one of the three standard numeric forms or, for the string type, the length and address of the string. Each of the four types is shown below:

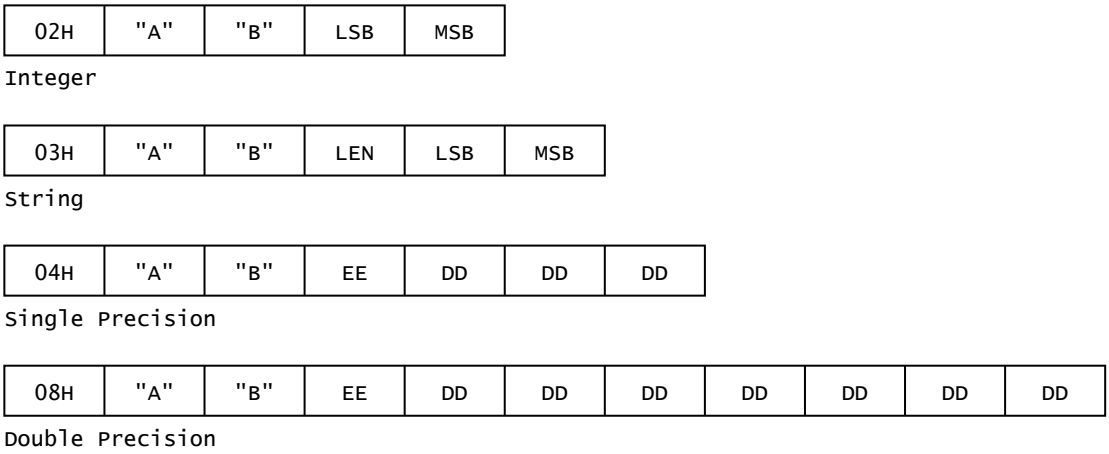


Figure 46: Simple Variables

NOFUNS is first checked to determine whether a user defined function is currently being evaluated. If so the search is carried out on the contents of PARM1 first of all, only if this fails will it move onto the main Variable Storage Area. A linear search method is used, the two name characters and type byte of each Variable in the storage area are compared to the reference characters and type until a match is found or the end of the storage area is reached. If the search is successful the routine terminates with the address of the first byte of the Variable contents in register pair DE. If the search is unsuccessful the Array Storage Area is moved upwards and the new Variable is added to the end of the existing ones and initialized to zero.

There are two exceptions to this automatic creation of a new Variable. If the search is being carried out by the "`VARPTR`" function, and this is determined by examining the return address, no Variable will be created. Instead the routine terminates with register pair DE set to zero (5F61H) causing a subsequent "`Illegal function call`" error. The second exception occurs when the search is being carried out by the Factor Evaluator, that is when the Variable is newly declared inside an expression. In this case DAC is zeroed for numeric types, and loaded with the address of a dummy zero length descriptor for a string type, thus returning a zero result (5FA7H). These actions are designed to prevent the Expression Evaluator creating a new Variable ("`VARPTR`") is the only function to take a Variable argument directly rather than via an expression and so requires separate protection). If this were not so then assignment to an Array, via the "`LET`" statement handler, would fail as any simple Variable created during expression evaluation would change the Array's address.

```
Address... 5FBAH
```

This is the Array search routine. There are four types of Array each composed of a header plus a number of elements. The first byte of the header contains the type code, the next two bytes the Array name and the next two the offset to the start of the following Array. This is followed by a single byte containing the dimensionality of the Array and the element count list. Each two byte element count contains the maximum number of elements per dimension. These are stored in reverse order with the first one corresponding to the last subscript. The contents of each Array element are identical to the

contents of the corresponding simple Variable. The integer Array AB%(3,4) is shown below with each element identified by its subscripts, high memory is towards the top of the page:

(0,4)	(1,4)	(2,4)	(3,4)
(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

02H	"A"	"B"	Offset		Dim	Count		Count	
			2DH	00H	02H	05H	00H	04H	00H

Figure 47: Integer Array

Each subscript is evaluated, converted to an integer (4755H) and pushed onto the Z80 stack until a closing parenthesis is found, it need not match the opening one. A linear search is then carried out on the Array Storage Area for a match with the two name characters and the type. If the search is successful DIMFLG is checked and a "`Redimensioned array`" error generated (405EH) if it shows a "`DIM`" statement to be active. Unless an "`ERASE`" statement is active, in which case the routine terminates with register pair BC pointing to the start of the Array (3297H), the dimensionality of the Array is then checked against the subscript count and a "`Subscript out of range`" error generated if they fail to match. Assuming these tests are passed control transfers to the element address computation point (607DH).

If the search is unsuccessful and an "`ERASE`" statement is active an "`Illegal function call`" error is generated (475AH), otherwise the new Array is added to the end of the existing Array Storage Area. Initialization of the new Array proceeds by storing the two name characters, the type code and the dimensionality (the subscript count) followed by the element count for each dimension. If DIMFLG shows a "`DIM`" statement to be active the element counts are determined by the subscripts. If the Array is being created by default, with a statement such as "`A(1,2,3)=5`" for example, a default value of eleven is used. As each element count is stored the total size of the Array is accumulated in register pair DE by successive multiplications (314AH) of the element counts and the element size (the Array type). After a check that this amount of memory is available (6267H) STREND is increased the new area is zeroed and the Array size is stored, in slightly modified form, immediately after the two name characters. Unless the Array is being created by default, in which case the element address must be computed, the routine then terminates.

This is the element address computation point of the Array search routine. The location of a particular element within an Array involves the multiplication (314AH) of subscripts, element counts and element sizes. As there are a variety of ways this could be done the actual method used is best illustrated with an example. The location of element (1,2,3) in a 4*5*6 Array would initially be computed as (((3*5)+2)*4)+1. This is then multiplied by the element size (type) and added to the Array base address to obtain the address of the required element. The computation method is an

optimized form which minimizes the number of steps needed, it is equivalent to evaluating (3*(4*5))+(2*4)+(1). The element address is returned in register pair DE.

```
Address... 60B1H
```

This is the "`PRINT USING`" statement handler. Control transfers here from the general "`PRINT`" statement handler after the applicable output device has been set up. Upon termination control passes back to the general "`PRINT`" statement exit point (4AFFH) to restore the normal video output. The format string is evaluated (4C65H) and the address and length of the string body obtained from the descriptor. The program text pointer is then temporarily saved. Each character of the format string is examined until one of the possible template characters is found. If the character does not belong in a template it is simply output via the OUTDO standard routine. Once the start of a template is found this is scanned along until a non-template character is found. Control then passes to the numeric output routine (6192H) or the string output routine (6211H).

In either case the program text pointer is restored to register pair HL and the next operand evaluated (4C64H). For numeric output the information gained from the template scan is passed to the numeric conversion routine (3426H) in registers A, B and C and the resulting string displayed (6678H). For string output the required character count is passed to the "`LEFT$`" statement handler (6868H) in register C and the resulting string displayed (667BH). For either type of output the program text and format string are then examined to determine whether there are any further characters. If no operands exist the handler terminates. If the format string has been exhausted then it is restarted from the beginning (60BFH), otherwise scanning continues from the current position for the next operand (60f6H).

```
Address... 6250H
```

This routine is used by the Interpreter Mainloop and the Variable search routine to move a block of memory upwards. A check is first made to ensure that sufficient memory exists (6267H) and then the block of memory is moved. The top source address is supplied in register pair BC and the top destination address in register pair HL. Copying stops when the contents of register pair BC equal those of register pair DE.

```
Address... 625EH
```

This routine is used to check that sufficient memory is available between the top of the Array Storage Area and the base of the Z80 stack. On entry register C contains the number of words the caller requires. If this would narrow the gap to less than two hundred bytes an "`Out of memory`" error is generated.

```
Address... 6286H
```

This is the "`NEW`" statement handler. TRCFLG, AUTFLG and PTRFLG are zeroed and the zero end link is placed at the start of the Program Text Area. VARTAB is set to point to the byte following the end link and control drops into the run-clear routine.

```
Address... 629AH
```

This routine is used by the "`NEW`", "`RUN`" and "`CLEAR`" statement handlers to initialize the Interpreter variables. All interrupts are cleared (636EH) and the default Variable types in DEFTBL set to double precision. RNDX is reset (2C24H) and ONEFLG, ONELIN and OLDTXT are zeroed. MEMSIZ is copied to FRETOP to clear the String Storage Area and DATPTR set to the start of the Program Text Area (63C9H). The contents of VARTAB are copied into ARYTAB and STREND, to clear any Variables, all the I/O buffers are closed (6C1CH) and NLONLY is reset. SAVSTK and the Z80 SP are reset from STKTOP and TEMPPT is reset to the start of TEMPST to clear any string descriptors. The printer is shut down (7304H) and output restored to the screen (4AFFH). Finally PRMLEN, NOFUNS, PRMLN2, FUNACT, PRMSTK and SUBFLG are zeroed and the routine terminates.

```
Address... 631BH
```

This routine is used by the "`DEVICE ON`" statement handlers to enable an interrupt source, the address of the relevant device's TRPTBL. status byte is supplied in register pair HL. Interrupts are enabled by setting bit 0 of the status byte. Bits 1 and 2 are then examined and, if the device has been stopped and an interrupt has occurred, ONGSBF is incremented (634FH) so that the Runloop will process it at the end of the statement. Finally bit 1 of the status byte is reset to release any existing stop condition.

```
Address... 632EH
```

This routine is used by the "`DEVICE OFF`" statement handlers to disable an interrupt source, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. Bits 0 and 2 are examined to determine whether an interrupt has occurred since the end of the last statement, if so ONGSBF is decremented (6362H) to prevent the Runloop from picking it up. The status byte is then zeroed.

```
Address... 6331H
```

This routine is used by the "`DEVICE STOP`" statement handlers to suspend processing of interrupts from an interrupt source, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. Bits 0 and 2 are examined to determine whether an interrupt has occurred since the end of the last statement, if so ONGSBF is decremented (6362H) to prevent the Runloop from picking it up. Bit 1 of the status byte is then set.

```
Address... 633EH
```

This routine is used by the "`RETURN`" statement handler to release the temporary stop condition imposed during interrupt driven BASIC subroutines, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. Bits 0, and 2 are examined to determine whether a stopped interrupt has occurred since the subroutine was first activated. If so ONGSBF is incremented (634FH) so that the Runloop will pick it up at the end of the statement. Bit 1 of the status byte is then reset. It should be noted that any "`DEVICE STOP`" Statement within an interrupt driven subroutine will therefore be ineffective.

```
Address... 6358H
```

This routine is used by the Runloop interrupt processor (6389H) to clear an interrupt prior to activating the BASIC subroutine, the address of the relevant device's TRPTBL status byte is supplied in register pair HL. ONGSBF is decremented and bit 2 of the status byte is reset.

```
Address... 636EH
```

This routine is used by the run-clear routine (629AH) to clear all interrupts. The seventy-eight bytes of TRPTBL and the ten bytes of FNKFLG are zeroed.

```
Address... 6389H
```

This is the Runloop interrupt processor. ONEFLG is first examined to determine whether an error condition currently exists. If so the routine terminates, no interrupts will be processed until the error clears. CURLIN is then examined and, if the Interpreter is in direct mode, the routine terminates. Assuming all is well a search is made of the twenty-six status bytes in TRPTBL to find the first active interrupt. Note that devices near the start of the table will consequently have a higher priority than those lower down. When the first active status byte is found, that is one with bits 0 and 2 set, the associated address is taken from TRPTBL and placed in register pair DE. The interrupt is then cleared (6358H) and the device stopped (6331H) before control transfers to the "`GOSUB`" handler (47CFH).

```
Address... 63C9H
```

This is the "`RESTORE`" statement handler. If no line number operand exists DATPTR is set to the start of the Program Storage Area. Otherwise the operand is collected (4769H), the program text searched to find the relevant line (4295H) and its address placed in DATPTR.

```
Address... 63E3H
```

This is the " `STOP` " statement handler. If further text exists in the statement control transfers to the " `STOP ON/OFF/STOP` " statement handler (77A5H). Otherwise register A is set to 01H and control drops into the " `END` " statement handler.

```
Address... 63EAH
```

This is the " `END` " statement handler. It is also used, with differing entry points, by the " `STOP` " statement and for CTRL- STOP and end of text program termination. ONEFLG is first zeroed and then, for the " `END` " statement only, all I/O buffers are closed (6C1CH). The current program text position is placed in SAVTXT and OLDTXT and the current line number in OLDLIN for use by any subsequent " `CONT` " statement. The printer is shut down (7304H), a CR LF issued to the screen (7323H) and register pair HL set to point to the " `Break` " message at 3FDCH. For the " `END` " statement and end of text cases control then transfers to the Mainloop " `OK` " point (411EH). For the CTRL-STOP case control transfers to the end of the error handler (40FDH) to display the " `Break` " message.

```
Address... 6424H
```

This is the " `CONT` " statement handler. Unless they are zero, in which case a " `Can't CONTINUE` " error is generated, the contents of OLDTXT are placed in register pair HL and those of OLDLIN in CURLIN. Control then returns to the Runloop to execute at the old program text position. A program cannot be continued after CTRL-STOP has been used to break from WITHIN a statement, via the CKCNTC standard routine, rather than from between statements.

```
Address... 6438H
```

This is the " `TRON` " statement handler, TRCFLG is simply made non-zero.

```
Address... 6439H
```

This is the " `TROFF` " statement handler, TRCFLG is simply made zero.

```
Address... 643EH
```

This is the " `SWAP` " statement handler. The first Variable is located (5EA4H) and its contents copied to SWPTMP. The location of this Variable and of the end of the Variable Storage Area are temporarily saved. The second Variable is then located (5EA4H) and its type compared with that of the first. If the types fail to match a " `Type mismatch` " error is generated (406DH). The current end of the Variable Storage Area is then compared with the old end and an " `Illegal function call` " error generated (475AH) if they differ. Finally the contents of the second Variable are copied to the location of the first Variable (2EF3H) and the contents of SWPTMP to the location of the second Variable (2EF3H).

The checks performed by the handler mean that the second Variable, if it is simple and not an Array, must always be in existence before a " `SWAP` " Statement is encountered or an error will be generated. The reason for this is that, supposing the first Variable was an Array, then the creation of a second (simple) Variable would move the Array Storage Area upwards invalidating its saved location. Note that the perfectly legal case of a simple first Variable and a newly created simple second Variable is also rejected.

```
Address... 6477H
```

This is the " `ERASE` " statement handler. SUBFLG is first set to 01H, to control the Variable search routine, and the Array located (5EA4H). All the following Arrays are moved downward and STREND set to its new, lower value. The program text is then checked and, if a comma follows, control transfers back to the start of the handler.

```
Address... 64A7H
```

This routine checks whether the character whose address is supplied in register pair HL is upper case alphabetic, if so it returns Flag NC.

```
Address... 64AFH
```

This is the " `CLEAR` " statement handler. If no operands are present control transfers to the run-clear routine (62A1H) to remove all current Variables. Otherwise the string space operand is evaluated (4756H) followed by the optional top of memory operand (542FH). The top of memory value is checked and an " `Illegal function call` " error generated (475AH) if it is less than 8000H or greater than F380H. The space required by the I/O buffers (267 bytes each) and the String Storage Area is subtracted from the top of memory value and an " `Out of memory` " error generated (6275H) if there is less than 160 bytes remaining to the base of the Variable Storage Area. Assuming all is well HIMEM, MEMSIZ and STKTOP are set to their new values and the remaining storage pointers reset via the run-clear routine (62A1H). The I/O buffer storage is re-allocated (7E6BH) and the handler terminates.

Unfortunately the computation of MEMSIZ and STKTOP, when a new top of memory is specified, is incorrect resulting in the top of the String Storage Area being set one byte too high. This can be seen with the following where an illegal string is accepted:

```
10 CLEAR 200,&HF380
20 A$=STRING$(201,"A")
30 PRINT FRE("")
```

Because there should be an extra DEC HL instruction at 64EBH the new values of MEMSIZ and STKTOP are initially set one byte too high. When the run-clear routine is called MEMSIZ is copied into FRETOP, the top of the String Storage Area, which results in this being one byte too high as well.

Although MEMSIZ and STKTOP are correctly recomputed when the file pointers are reset, FRETOP is left with its incorrect value. When the "`FRE`" statement is executed in line thirty, and string garbage collection initiated, FRETOP is restored to its correct value but, because the string overflows the String Storage Area by one byte, the amount of free space displayed is -1 byte. To correctly set all the system pointers any alteration of the top of memory should be followed immediately by another "`CLEAR`" statement with no operands.

```
Address... 6520H
```

This routine computes the difference between the contents of register pairs HL and DE. It is a duplicate of the short section of code from 64ECH to 64F1H and is completely unused.

```
Address... 6527H
```

This is the "`NEXT`" statement handler. Assuming further text is present in the statement the loop Variable is located (5EA4H), otherwise a default address of zero is taken. The stack is then searched for the corresponding "`FOR`" parameter block (3FE2H). If no parameter block is found, or if a "`GOSUB`" parameter block is found first, a "`NEXT without FOR`" error is generated (405BH). Assuming the parameter block is found the intervening section of stack, together with any "`FOR`" blocks it may contain, is discarded. The loop Variable type is then taken from the parameter block and examined to determine the precision required during subsequent operations.

The STEP value is taken from the parameter block and added (3172H, 324EH or 2697H) to the current contents of the loop Variable which is then updated. The new value is compared (2F4DH, 2F21H or 2F5CH) with the termination value from the parameter block to determine whether the loop has terminated (65B6H). The loop will terminate for a positive STEP if the new loop value is GREATER than the termination value. The loop will terminate for a negative step if the new loop value is LESS than the termination value. If the loop has not terminated the original program text position and line number are taken from the parameter block and control transfers to the Runloop (45FDH). If the loop has terminated the parameter block is discarded from the stack and, unless further program text is present in which control transfers back to the start of the handler, control transfers to the Runloop to execute the next statement (4601H).

```
Address... 65C8H
```

This routine is used by the Expression Evaluator to find the relation (<>=) between two string operands. The address of the first string descriptor is supplied on the Z80 stack and the address of the second in DAC. The result is returned in register A and the flags as for the numeric relation routines:

```
String 1=String 2 ... A=00H, Flag Z,NC
String 1<String 2 ... A=01H, Flag NZ,NC
```

```
String 1>String 2 ... A=FFH, Flag NZ,C
```

Comparison commences at the first character of each string and continues until the two characters differ or one of the strings is exhausted. Control then returns to the Expression Evaluator (4F57H) to place the true or false numeric result in DAC.

```
Address... 65F5H
```

This routine is used by the Factor Evaluator to apply the " `OCT$` " function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (371EH) and then the result string is created (6607H).

```
Address... 65FAH
```

This routine is used by the Factor Evaluator to apply the " `HEX$` " function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (3722H) and then the result string is created (6607H).

```
Address... 65FFH
```

This routine is used by the Factor Evaluator to apply the " `BIN$` " function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (371AH) and then the result string is created (6607H).

```
Address... 6604H
```

This routine is used by the Factor Evaluator to apply the " `STR$` " function to an operand contained in DAC. The number is first converted to textual form in FBUFFR (3425H) then analyzed to determine its length and address (6635H). After checking that sufficient space is available (668EH) the string is copied to the String Storage Area (67C7H) and the result descriptor created (6654H).

```
Address... 6627H
```

This routine first checks that there is sufficient space in the String Storage Area for the string whose length is supplied in register A (668EH). The string length and the address where the string will be placed in the String Storage Area are then copied to DSCTMP.

```
Address... 6636H
```

This routine is used by the Factor Evaluator to analyze the character string whose address is supplied in register pair HL. The character string is scanned until a terminating character (00H or ") is found. The length and starting address are then placed in DSCTMP (662AH) and control drops into the descriptor creation routine.

```
Address... 6654H
```

This routine is used by the string functions to create a result descriptor. The descriptor is copied from DSCTMP to the next available position in TEMPST and its address placed in DAC. Unless TEMPST is full, in which case a "`String formula too complex`" error is generated, TEMPPT is increased by three bytes and the routine terminates.

```
Address... 6678H
```

This routine displays the message, or string, whose address is supplied in register pair HL. The string is analyzed (6635H) and its storage freed (67D3H). Successive characters are then taken from the string and displayed, via the OUTDO standard routine, until the string is exhausted.

```
Address... 668EH
```

This routine checks that there is room in the String Storage Area to add the string whose length is supplied in register A. On exit register pair DE points to the starting address in the String Storage Area where the string should be placed. The length of the string is first subtracted from the current free location contained in FRETOP. This is then compared with STKTOP, the lowest allowable location for string storage, to determine whether there is space for the string. If so FRETOP is updated with the new position and the routine terminates. If there is insufficient space for the string then garbage collection is initiated (66B6H) to try and eliminate any dead strings. If, after garbage collection, there is still not enough space an "`Out of string space`" error is generated.

```
Address... 66B6H
```

This is the string garbage collector, its function is to eliminate any dead strings from the String Storage Area. The basic problem with string Variables, as opposed to numeric ones, is that their lengths vary. If string bodies were stored with their Variables in the Variable Storage Area even such apparently simple statements as A$=A$+"X" would require the movement of thousands of bytes of memory and slow execution speeds dramatically. The method used by the Interpreter to overcome this problem is to keep the string bodies separate from the Variables. Thus strings are kept in the String Storage Area and each Variable holds a three byte descriptor containing the length and address of the associated string. Whenever a string is assigned to a Variable it is simply added to the heap of existing strings in the String Storage Area and the Variable's descriptor changed. No attempt

is made to eliminate any previous string belonging to the Variable, by restructuring the heap, as this would wipe out any throughput gains.

If sufficient Variable assignments are made it is inevitable that the String Storage Area will fill up. In a typical program many of these strings will be unused, that is the result of previous assignments. Garbage collection is the process whereby these dead strings are removed. Every string Variable in memory, including Arrays and the local Variables present during evaluation of user defined functions, is examined until the one is found whose string is stored highest in the heap. This string is then moved to the top of the String Storage Area and the Variable contents modified to point to the new location. The owner of the next highest string is then found and the process repeated until every string belonging to a Variable has been compacted.

If a large number of Variables are present garbage collection may take an appreciable time. The process can be seen at work with the following program which repeatedly assigns the string " `AAAA` " to each element of the Array A$. The program will run at full speed for the first two hundred and fifty assignments and then pause to eliminate the fifty dead strings. A further fifty assignments can then be made before a further garbage collection is required:

```
10 CLEAR 1000
20 DIM A$(200)
30 FOR N=0 TO 200
40 A$(N)=STRING$(4,"A")
50 PRINT".";
60 NEXT N
70 GOTO 30
```

The String Storage Area is also used to hold the intermediate strings produced during expression evaluation. Because so many string functions take multiple arguments, " `MID$` " takes three for example, the management of intermediate results is a major problem. To deal with it a standardized approach to string results is taken throughout the Interpreter. A producer of a string simply adds the string body to the heap in the String Storage Area, adds the descriptor to the descriptor heap in TEMPST and places the address of the descriptor in DAC. It is up to the user of the result to free this storage (67D0H) once it has processed the string. This rule applies to all parts of the system, from the individual function handlers back through the Expression Evaluator to the statement handlers, with only two exceptions.

The first exception occurs when the Factor Evaluator finds an explicitly stated string, such as " `SOMETHING` " in the program text. In this case it is not necessary to copy the string to the String Storage Area as the original will suffice.

The second exception occurs when the Factor Evaluator finds a reference to a Variable. In this case it is not necessary to place a copy of the descriptor in TEMPST as one already exists inside the Variable.

```
Address... 6787H
```

This routine is used by the Expression Evaluator to concatenate two string operands. Control transfers here when a "+" token is found following a string operand so the first action taken is to fetch the second string operand via the Factor Evaluator (4DC7H). The lengths are then taken from both string descriptors and added together to check the length of the combined string. If this is greater than two hundred and fifty-five characters a "`String too long`" error is generated. After checking that space is available in the String Storage Area (6627H) the storage of both operands is freed (67D6H). The first string is then copied to the String Storage Area (67BFH) and followed by the second one (67BFH). The result descriptor is created (6654H) and control transfers back to the Expression Evaluator (4C73H)'

```
Address... 67D0H
```

This routine frees any storage occupied by the string whose descriptor address is contained in DAC. The address of the descriptor is taken from DAC and examined to determine whether it is that of the last descriptor in TEMPST (67EEH), if not the routine terminates. Otherwise TEMPPT is reduced by three bytes clearing this descriptor from TEMPST. The address of the string body is then taken from the descriptor and compared with FRETOP to see if this is the lowest string in the String Storage Area, if not the routine terminates. Otherwise the length of the string is added to FRETOP, which is then updated with this new value, freeing the storage occupied by the string body.

```
Address... 67FFH
```

This routine is used by the Factor Evaluator to apply the "`LEN`" function to an operand contained in DAC. The operand's storage is freed (67D0H) and the string length taken from the descriptor and placed in DAC as an integer (4FCFH).

```
Address... 680BH
```

This routine is used by the Factor Evaluator to apply the "`ASC`" function to an operand contained in DAC. The operand's storage is freed and the string length examined (6803H), if it is zero an "`Illegal function call`" error is generated (475AH). Otherwise the first character is. taken from the string and placed in DAC as an integer (4FCFH).

```
Address... 681BH
```

This routine is used by the Factor Evaluator to apply the "`CHR$`" function to an operand contained in DAC. After checking that sufficient space is available (6625H) the operand is converted to a single byte integer (521FH). This character is then placed in the String Storage Area and the result descriptor created (6654H).

```
Address... 6829H
```

This routine is used by the Factor Evaluator to apply the " `STRING$` " function. After checking for the open parenthesis character the length operand is evaluated and placed in register E (521CH). The second operand is then evaluated (4C64H). If it is numeric it is converted to a single byte integer (521FH) and placed in register A. If it is a string the first character is taken from it and placed in register A (680FH). Control then drops into the " `SPACE$` " function to create the result string.

```
Address... 6848H
```

This routine is used by the Factor Evaluator to apply the " `SPACE$` " function to an operand contained in DAC. The operand is first converted to a single byte integer in register E (521FH). After checking that sufficient space is available (6627H) the required number of spaces are copied to the String Storage Area and the result descriptor created (6654H).

```
Address... 6861H
```

This routine is used by the Factor Evaluator to apply the " `LEFT$` " function. The first operand's descriptor address and the integer second operand are supplied on the Z80 stack. The slice size is taken from the stack (68E3H) and compared to the source string length. If the source string length is less than the slice size it replaces it as the length to extract. After checking that sufficient space is available (668EH) the required number of characters are copied from the start of the source string to the String Storage Area (67C7H). The source string's storage is then freed (67D7H) and the result descriptor created (6654H).

```
Address... 6891H
```

This routine is used by the Factor Evaluator to apply the " `RIGHT$` " function. The first operand's descriptor address and the integer second operand are supplied on the Z80 stack. The slice size is taken from the stack (68E3H) and subtracted from the source string length to determine the slice starting position. Control then transfers to the " `LEFT$` " routine to extract the slice (6865H).

```
Address... 689AH
```

This routine is used by the Factor Evaluator to apply the " `MID$` " function. The first operand's descriptor address and the integer second operand are supplied on the Z80 stack. The starting position is taken from the stack (68E6H) and checked, if it is zero an " `Illegal function call` " error is generated (475AH). The optional slice size is then evaluated (69E4H) and control transfers to the " `LEFT$` " routine to extract the slice (6869H).

```
Address... 68BBH
```

This routine is used by the Factor Evaluator to apply the " `VAL` " function to an operand contained in DAC. The string length is taken from the descriptor (6803H) and checked, if it is zero it is placed in DAC as an integer (4FCFH). The length is then added to the starting address of the string body to give the location of the character immediately following it. This is temporarily replaced with a zero byte and the string is converted to numeric form in DAC (3299H). The original character is then restored and the routine terminates. The temporary zero byte delimiter is necessary because strings are packed together in the String Storage Area, without it the numeric converter would run on into succeeding strings.

```
Address... 68E3H
```

This routine is used by the " `LEFT$` ", " `MID$` " and " `RIGHT$` " function handlers to check that the next program text character is ")" and then to pop an operand from the Z80 stack into register pair DE.

```
Address... 68EBH
```

This routine is used by the Factor Evaluator to apply the " `INSTR` " function. The first operand, which may be the starting position or the source string, is evaluated (4C62H) and its type tested. If it is the source string a default starting position of one is taken. If it is the starting position operand its value is checked and the source string operand evaluated (4C64H). The pattern string is then evaluated (4C64H) and the storage of both operands freed (67D0H). The length of the pattern string is checked and, if zero, the starting position is placed in DAC (4FCFH). The pattern string is then checked against successive characters from the source string, commencing at the starting position, until a match is found or the source string is exhausted. With a successful search the character position of the substring is placed in DAC as an integer (4FCFH), otherwise a zero result is returned.

```
Address... 696EH
```

This is the " `MID$` " statement handler. After checking for the open parenthesis character the destination Variable is located (5EA4H) and checked to ensure that it is a string type (3058H). The address of the string body is then taken from the Variable and examined to determine whether it is inside the Program Text Area, as would be the case for an explicitly stated string. If this is the case the string body is copied to the String Storage Area (6611H) and a new descriptor copied to the Variable (2EF3H). This is done to avoid modifying the program text. The starting position is then evaluated (521CH) and checked, if it is zero an " `Illegal function call` " error is generated (475AH). The optional slice length operand is evaluated (69E4H) followed by the replacement string (4C5FH) whose storage is then freed (67D0H). Characters are then copied from the replacement string to the destination string until either the slice length is completed or the replacement string is exhausted.

```
Address... 69E4H
```

This routine is used by various string functions to evaluate an optional operand (521CH) and return the result in register E. If no operand is present a default value of 255 is returned.

```
Address... 69F2H
```

This routine is used by the Factor Evaluator to apply the "`FRE`" function to an operand contained in DAC. If the operand is numeric the single precision difference between the Z80 Stack Pointer and the contents of STREND is placed in DAC (4FC1H). If the operand is a string type its storage is freed (67D3H) and garbage collection initiated (66B6H). The single precision difference between the contents of FRETOP and those of STKTOP is then placed in DAC (4FC1H).

```
Address... 6A0EH
```

This routine is used by the file I/O handlers to analyze a filespec such as "`A:FILENAME.BAS`". The filespec consists of three parts, the device, the filename and the type extension. On entry register pair HL points to the start of the filespec in the program text. On exit register D holds the device code, the filename is in positions zero to seven of FILNAM and the type extension in positions eight to ten. Any unused positions are filled with spaces.

The filespec string is evaluated (4C64H) and its storage freed (67D0H), if the string is of zero length a "`Bad file name`" error is generated (6E6BH). The device name is parsed (6F15H) and successive characters taken from the filespec and placed in FILNAM until the string is exhausted, a "." character is found or FILNAM is full. A "`Bad file name`" error is generated (6E6BH) if the filespec contains any control characters, that is those whose value is smaller than 20H. If the filespec contains a type extension a "`Bad file name`" error is generated (6E6BH) if it is longer than three characters or if the filename is longer than eight characters. If no type extension is present the filename may be any length, extra characters are simply ignored.

```
Address... 6A6DH
```

This routine is used by the file I/O handlers to locate the I/O buffer FCB whose number is supplied in register A. The buffer number is first checked against MAXFIL and a "`Bad file number`" error generated (6E7DH) if it is too large. Otherwise the required address is taken from the file pointer block and placed in register pair HL and the buffer's mode taken from byte 0 of the FCB and placed in register A.

```
Address... 6A9EH
```

This routine is used by the file I/O handlers to evaluate an I/O buffer number and to locate its FCB. Any "#" character is skipped (4666H) and the buffer number evaluated (521CH). The FCB is located

(6A6DH) and a " `File not open` " error generated (6E77H) if the buffer mode byte is zero. Otherwise the FCB address is placed in PTRFIL to redirect the Interpreter's output.

```
Address... 6AB7H
```

This is the " `OPEN` " statement handler. The filespec is analyzed (6A0EH) and any following mode converted to the corresponding mode byte, these are: " `FOR INPUT` " (01H), " `FOR OUTPUT` " (02H) and " `FOR APPEND` " (08H). If no mode is explicitly stated random mode (04H) is assumed. The " `AS` " characters are checked and the buffer number evaluated (521CH), if this is zero a " `Bad file number` " error is generated (6E7DH). The FCB is then located (6A6DH) and a " `File already open` " error generated (6E6EH) if the buffer's mode byte is anything other than zero. The device code is placed in byte 4 of the FCB, the open function dispatched (6F8FH) and the Interpreter's output reset to the screen (4AFFH).

```
Address... 6B24H
```

This routine is used by the file I/O handlers to close the I/O buffer whose number is supplied in register A. The FCB is located (6A6DH) and, provided the buffer is in use, the close function dispatched (6F8FH) and the buffer filled with zeroes (6CEAH). PTRFIL and the FCB mode byte are then zeroed to reset the Interpreter's output to the screen.

```
Address... 6B5BH
```

This is the " `LOAD` ", " `MERGE` " and " `RUN filespec` " statement handler. The filespec is analyzed (6A0EH) and then, for " `LOAD` " and " `RUN` " only, the program text examined to determine whether the auto-run " `R` " option is specified. I/O buffer 0 is opened for input (6AFAH) and the first byte of FILNAM set to FFH if auto-run is required. For " `LOAD` " and " `RUN` " only any program text is then cleared via the " `NEW` " statement handler (6287H). As this will reset the Interpreter's output to the screen the buffer FCB is again located and placed in PTRFIL (6AAAH). Control then transfers directly to the Interpreter Mainloop (4134H) for the program text to be loaded as if typed from the keyboard. Note that no error checking of any sort is carried out on the data read.

```
Address... 6BA3H
```

This is the " `SAVE` " statement handler. The filespec is analyzed (6A0EH) and the program text examined to determine whether the ASCII " `A` " suffix is present. This is only relevant under Disk BASIC, it makes no difference on a standard MSX machine. I/O buffer 0 is opened for output (6AFAH) and control transfers to the " `LIST` " statement handler (522EH) to output the program text. Note that no error checking information of any sort accompanies the text.

```
Address... 6BDAH
```

This routine is used by the file I/O handlers to return the device code for the currently active I/O buffer. The FCB address is taken from PTRFIL then the device code taken from byte 4 of the FCB and placed in register A.

```
Address... 6BE7H
```

This routine is used by the file I/O handlers to perform an operation on a number of I/O buffers. The address of the relevant routine is supplied in register pair BC and the buffer count in register A. For example if register pair BC contained 6B24H and register A contained 03H buffers 3, 2, 1 and 0 would be closed. The routine has a slightly different function if it is entered with FLAG NZ. In this case the I/O buffer numbers are taken sequentially from the program text and evaluated (521CH) before the operation is performed, a typical case might be "#1,#2".

```
Address... 6C14H
```

This is the " `CLOSE` " statement handler. Register pair BC is set to 6B24H, register A is loaded with the contents of MAXFIL and the required number of buffers closed (6BE7H).

```
Address... 6C1CH
```

This routine is used by the file I/O handlers to close every I/O buffer. Register pair BC is set to 6B24H, register A is loaded with the contents of MAXFIL and all buffers closed (6BE7H).

```
Address... 6C2AH
```

This is the " `LFILES` " statement handler. PRTFLG is made non- zero, to direct output to the printer, and control drops into the " `FILES` " statement handler.

```
Address... 6C2FH
```

This is the " `FILES` " statement handler, an " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 6C35H
```

Control transfers here from the general " `PUT` " and " `GET` " handlers (7758H) when the program text contains anything other than a " `SPRITE` " token. A " `Sequential I/O only` " error is generated (6E86H)

on a standard MSX machine.

```
Address... 6C48H
```

This routine is used by the file I/O handlers to sequentially output the character supplied in register A. The character is placed in register C and the sequential output function dispatched (6F8FH).

```
Address... 6C71H
```

This routine is used by the file I/O handlers to sequentially input a single character. The sequential input function is dispatched (6F8FH) and the character returned in register A, FLAG C indicates an EOF (End Of File) condition.

```
Address... 6C87H
```

This routine is used by the Factor Evaluator to apply the "`INPUT$`" function. The program text is checked for the "$" and "(" characters and the length operand evaluated (521CH). If an I/O buffer number is present it is evaluated, the FCB located (6A9EH) and the mode byte examined. An "`Input past end`" error is generated (6E83H) if the buffer is not in input or random mode. After checking that sufficient space is available (6627H) the required number of characters are sequentially input (6C71H), or collected via the CHGET standard routine, and copied to the String Storage Area. Finally the result descriptor is created (6654H).

```
Address... 6CEAH
```

This routine is used by the file I/O handlers to fill the buffer whose FCB address is contained in PTRFIL with two hundred and fifty-six zeroes.

```
Address... 6CFBH
```

This routine is used by the file I/O handlers to return, in register pair HL, the starting address of the buffer whose FCB address is contained in PTRFIL. This just involves adding nine to the FCB address.

```
Address... 6D03H
```

This routine is used by the Factor Evaluator to apply the "`LOC`" function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the LOC function dispatched (6F8FH). An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 6D14H
```

This routine is used by the Factor Evaluator to apply the " `LOF` " function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the LOF function dispatched (6F8FH). An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 6D25H
```

This routine is used by the Factor Evaluator to apply the " `EOF` " function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the EOF function dispatched (6F8FH).

```
Address... 6D39H
```

This routine is used by the Factor Evaluator to apply the " `FPOS` " function to the I/O buffer whose number is contained in DAC. The FCB is located (6A6AH) and the `FPOS` function dispatched (6F8FH). An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 6D48H
```

Control transfers to this routine when the Interpreter Mainloop encounters a direct statement, that is one with no line number. The ISFLIO standard routine is first used to determine whether a " `LOAD` " statement is active. If input is coming from the keyboard control transfers to the Runloop execution point (4640H) to execute the statement. If input is coming from the cassette buffer 0 is closed (6B24H) and a " `Direct statement in file` " error generated (6E71H). This could happen on a standard MSX machine either through a cassette error or by attempting to load a text file with no line numbers.

```
Address... 6D57H
```

This routine is used by the " `INPUT` ", " `LINE INPUT` " and " `PRINT` " statement handlers to check for the presence of a "#" character in the program text. If one is found the I/O buffer number is evaluated (521BH), the FCB located and its address placed in PTRFIL (6AAAH). The mode byte of the FCB is then compared with the mode number supplied by the statement handler in register C, if they do not match a " `Bad file number` " error is generated (6E7DH). With " `PRINT` " the allowable modes are output, random and append. With " `INPUT` " and " `LINE INPUT` " the allowable modes are input and random. Note that on a standard MSX machine not all these modes are supported at lower levels. Some sort of error will consequently be generated at a later stage for illegal modes.

```
Address... 6D83H
```

This routine is used by the "`INPUT`" statement handler to input a string from an I/O buffer. A return is first set up to the "`READ/INPUT`" statement handler (4BF1H). The characters which delimit the input string, comma and space for a numeric Variable and comma only for a string Variable, are placed in registers D and E and control transfers to the "`LINE INPUT`" routine (6DA3H).

```
Address... 6D8FH
```

This is the "`LINE INPUT`" statement handler when input is from an I/O buffer. The buffer number is evaluated, the FCB located and the mode checked (6D55H). The Variable to assign to is then located (5EA4H) and its type checked to ensure it is a string type (3058H). A return is set up to the "`LET`" statement handler (487BH) to perform the assignment and the input string collected.

Characters are sequentially input (6C71H) and placed in BUF until the correct delimiter is found, EOF is reached or BUF fills up (6E41H). When the terminating condition is reached and assignment is to a numeric Variable the string is converted to numeric form in DAC (3299H). When assignment is to a string Variable the string is analyzed and the result descriptor created (6638H).

For "`LINE INPUT`" all characters are accepted until a CR code is reached. Note that if this CR code is preceded by a LF code then it will not function as a delimiter but will merely be accepted as part of the string. For "`INPUT`" to a numeric Variable leading spaces are stripped then characters accepted until a CR code, a space or a comma is reached. Note that as for "`LINE INPUT`" a CR code will not function as a delimiter when preceded by a LF code. In this case however the CR code will not be placed in BUF but ignored. For "`INPUT`" to a string Variable leading spaces are stripped then characters accepted until a CR or comma is reached. Note that as for "`LINE INPUT`" a CR code will not function as a delimiter when preceded by a LF code. In this case however neither code will be placed in BUF both are ignored. An alternative mode is entered when the first character read, after any spaces, is a double quote character. In this case all characters will be accepted, and stored in BUF, until another double quote delimiter is read.

Once the input string has been accepted the terminating delimiter is examined to see if any special action is required with respect to trailing characters. If the input string was delimited by a double quote character or a space then any succeeding spaces will be read in and ignored until a non-space character is found. If this character is a comma or CR code then it is accepted and ignored. Otherwise a putback function is dispatched (6F8FH) to return the character to the I/O buffer. If the input string was delimited by a CR code then the next character is read in and checked. If this is a LF code it will be accepted but ignored. If it is not a LF code then a putback function is dispatched (6F8FH) to return the character to the I/O buffer.

```
Address... 6E6BH
```

This is a group of ten file I/O related error generators. Register E is loaded with the relevant error code and control transfers to the error handler (406FH):

ADDRESS	ERROR
6E6BH	Bad file name
6E6EH	File already open
6E71H	Direct statement in file
6E74H	File not found
6E77H	File not open
6E7AH	Field overflow
6E7DH	Bad file number
6E80H	Internal error
6E83H	Input past end
6E86H	Sequential I/O only

```
Address... 6E92H
```

This is the "`BSAVE`" statement handler. The filespec is analyzed (6A0EH) and the start address evaluated (6F0BH). The stop address is then evaluated (6F0BH) and placed in SAVEND followed by the optional entry address (6F0BH) which is placed in SAVENT. If no entry address exists the start address is taken instead. The device code is checked to ensure that it is CAS, if not a "`Bad file name`" error is generated (6E6BH), and the data written to cassette (6FD7H). Note that no buffering is involved, data is written directly to the cassette, and no error checking information accompanies the data.

```
Address... 6EC6H
```

This is the "`BLOAD`" statement handler. The filespec is analyzed (6A0EH) and RUNBNF made non-zero if the auto-run "`R`" option is present in the program text. The optional load offset, with a default value of zero, is then evaluated (6F0BH) and the device code checked to ensure that it is CAS, if not a "`Bad file name`" error is generated (6E6BH). Data is then read directly from cassette (7014H), as with "`BSAVE`" no buffering or error checking is involved.

```
Address... 6EF4H
```

Control transfers to this routine when the "`BLOAD`" statement handler has completed loading data into memory. If RUNBNF is zero buffer 0 is closed (6B24H) and control returns to the Runloop. Otherwise buffer 0 is closed (6B24H), a return address of 6CF3H is set up (this routine just pops the program text pointer back into register pair HL and returns to the Runloop) and control transfers to the address contained in SAVENT.

```
Address... 6F0BH
```

This routine is used by the "`BLOAD`" and "`BSAVE`" handlers to evaluate an address operand, the result is returned in register pair DE. The operand is evaluated (4C64H) then converted to an integer (5439H).

```
Address... 6F15H
```

This routine is used by the filespec analyzer to parse a device name such as "`CAS:`". On entry register pair HL points to the start of the filespec string and register E contains its length. If no device name is present the default device code (CAS=FFH) is returned in register A with FLAG Z. If a legal device name is present its code is returned in register A with FLAG NZ.

The filespec is examined until a ":" character is found then the name compared with each of the legal device names in the device table at 6F76H. If a match is found the device code is taken from the table and returned in register A. If no match is found control transfers to the external ROM search routine (55F8H). Note that any lower case characters are turned to upper case for comparison purposes. Thus crt and CRT, for example, are the same device.

```
Address... 6F76H
```

This table is used by the device name parser, it contains the four device names and codes available on a standard MSX machine:

```
CAS ... FFH  LPT ... FEH  CRT ... FDH  GRP ... FCH
```

```
Address... 6F87H
```

This table is used by the function dispatcher (6F8FH), it contains the address of the function decoding table for each of the four standard MSX devices:

```
CAS ... 71C7H  LPT ... 72A6H  CRT ... 71A2H  GRP ... 7182H
```

```
Address... 6F8FH
```

This is the file I/O function dispatcher. In conjunction with the Interpreter's buffer structure it provides a consistent, device independent method of inputting or outputting data. The required function code is supplied in register A and the address of the buffer FCB in register pair HL.

The device code is taken from byte 4 of the FCB and examined to determine whether it is one of the four standard devices, if not control transfers to the external ROM function dispatcher (564AH). Otherwise the address of the device's function decoding table is taken from the table at 6F87H, the required function's address taken from it and control transferred to the relevant function handler.

```
Address... 6FB7H
```

This is the "`CSAVE`" statement handler. The filename is evaluated (7098H) followed by the optional baud rate operand (7A2DH). The identification block is then written to cassette (7125H) with a filetype byte of D3H. The contents of the Program Text Area are written directly to cassette as a single data block (713EH). Note that no error checking information accompanies the data.

```
Address... 6FD7H
```

Control transfers to this routine from the "`BSAVE`" statement handler to write a block of memory to cassette. The identification block is first written to cassette (7125H) with a filetype byte of D0H. The motor is then turned on and a short header written to cassette (72F8H) The starting address is popped from the Z80 stack and written to cassette LSB first, MSB second (7003H). The stop address is taken from SAVEND and written to cassette LSB first, MSB second (7003H). The entry address is taken from SAVENT and written to cassette LSB first, MSB second (7003H). The required area of memory is then written to cassette one byte at a time (72DEH) and the cassette motor turned off via the TAPOOF standard routine. Note that no error checking information accompanies the data.

```
Address... 7003H
```

This routine writes the contents of register pair HL to cassette with register L first (72DEH) and register H second (72DEH).

```
Address... 700BH
```

This routine reads two bytes from cassette and places the first in register L (72D4H), the second in register H (72D4H).

```
Address... 7014H
```

Control transfers to this routine from the "`BLOAD`" statement handler to load data from the cassette into memory. The cassette is read until an identification block with a file type of D0H and the correct filename is found (70B8H). The data block header is then located on the cassette (72E9H). The offset value is popped from the Z80 stack and added to the start address from the cassette (700BH). The stop address is read from cassette (700BH) and the offset added to this as well. The entry address is read from cassette (700BH) and placed in SAVENT in case auto-run is required. Successive data bytes

are then read from cassette (72D4H) and placed in memory, at the start address initially, until the stop address is reached. Finally the motor is turned off via the TAPIOF standard routine and control transfers to the " `BLOAD` " termination point (6EF4H).

```
Address... 703FH
```

This is the " `CLOAD` " and " `CLOAD?` " statement handler. The program text is first checked for a trailing " `PRINT` " token (91H) which is how the " `?` " character is tokenized. The filename is then evaluated (708CH) and the cassette read until an identification block with a filetype of D3H and the correct filename is found (70B8H). For " `CLOAD` " a " `NEW` " operation is then performed (6287H) to erase the current program text. For " `CLOAD?` " all pointers in the Program Text Area are converted to line numbers (54EAH) to match the cassette data.

The data block header is located on the cassette and successive data bytes read from cassette and placed in memory or compared with the current memory contents (715DH). When the data block has been completely read the message " `OK` " is displayed (6678H) and control transfers directly to the end of the Interpreter Mainloop (4237H) to reset the Variable storage pointers. For " `CLOAD?` " reading of the data block will terminate if the cassette byte is not the same as the program text byte in memory. If the address where this occurred is above the end of the Program Text Area then the handler terminates with an " `OK` " message as before. Otherwise a " `Verify error` " is generated.

```
Address... 708CH
```

This routine is used by the " `CLOAD` " and " `CSAVE` " statement handlers to evaluate a filename in the program text. The two handlers use different entry points so that a null filename is allowed for " `CLOAD` " but not for " `CSAVE` ". The filename string is evaluated (4C64H), its storage freed (680FH) and the first six characters copied to FILNAM. If the filename is longer than six characters the excess is ignored. If the filename is shorter than six characters then FILNAM is padded with spaces.

```
Address... 70B8H
```

This routine is used by the " `CLOAD` " and " `BLOAD` " statement handlers and for the dispatcher open function (when the device is CAS and the mode is input) to locate an identification block on the cassette. On entry the filename is in FILNAM and the file type in register C, D3H for a tokenized BASIC (`CLOAD`) file, D0H for a binary (`BLOAD`) file and EAH for an ASCII (`LOAD` or data) file.

The cassette motor is turned on and the cassette read until a header is found (72E9H). Each identification block is prefixed by ten file type characters so successive characters are read from cassette (72D4H) and compared to the required file type. If the file type characters do not match control transfers back to the start of the routine to find the next header. Otherwise the next six characters are read in (72D4H) and placed in FILNAM. If FILNAM is full of spaces no filename match is attempted and the identification block has been found. Otherwise the contents of FILNAM and

FILNM2 are compared to determine whether this is the required file. If the match is unsuccessful, and the Interpreter is in direct mode, the message " `Skip:` " is displayed (710DH) followed by the filename. Control then transfers back to the start of the routine to try the next header. If the match is successful, and the Interpreter is in direct mode, the message " `Found:` " is displayed (710DH) followed by the filename and the routine terminates.

```
Address... 70FFH
```

This is the plain text message " `Found:` " terminated by a zero byte.

```
Address... 7106H
```

This is the plain text message " `Skip :` " terminated by a zero byte.

```
Address... 710DH
```

Unless CURLIN shows the Interpreter to be in program mode this routine first displays (6678H) the message whose address is supplied in register pair HL, followed by the six characters contained in FILNM2.

```
Address... 7125H
```

This routine is used by the " `CSAVE` " and " `BSAVE` " statement handlers and for the dispatcher open function (when the device is CAS and the mode is output) to write an identification block to cassette. On entry the filename is in FILNAM and the filetype in register A, D3H for a tokenized BASIC (`CSAVE`) file, D0H for a binary (`BSAVE`) file and EAH for an ASCII (`SAVE` or data) file. The cassette motor is turned on and a long header written to cassette (72F8H) The filetype byte is then written to cassette (72DEH) ten times followed by the first six characters from FILNAM (72DEH). The cassette motor is turned off via the TAPOOF standard routine and the routine terminates.

```
Address... 713EH
```

This routine is used by the " `CSAVE` " statement handler to write the Program Text Area to cassette as a single data block. All pointers in the program text are converted back to line numbers (54EAH) to make the text address independent. The cassette motor is turned on and a short header written to cassette (72F8H) The entire Program Text Area is then written to cassette a byte at a time (72DEH) and followed with seven zero bytes (72DEH) as a terminator. The cassette motor is then turned off via the TAPOOF standard routine and the routine terminates.

```
Address... 715DH
```

This routine is used by the "`CLOAD`" and "`CLOAD?`" statement handlers to read a single data block into the Program Text Area or to compare it with the current contents. On entry register A contains a flag to distinguish between the two statements, 00H for "`CLOAD`" and FFH for "`CLOAD?`". The cassette motor is turned on and the first header located (72E9H). Successive characters are read from cassette (72D4H) and placed in the Program Text Area or compared with the current contents. If the current statement is "`CLOAD?`" the routine will terminate with FLAG NZ if the cassette character is not the same as the memory character. Otherwise data will be read until ten successive zeroes are found. This sequence of zeroes is composed of the last program line end of line character, the end link and the seven terminator zeroes added by "`CSAVE`". Note that the routine will probably terminate during this sequence, when used by "`CLOAD?`", as memory comparison is still in progress. This accounts for the rather peculiar coding of the "`CLOAD?`" handler terminating conditions.

```
Address... 7182H
```

This table is used by the dispatcher when decoding function codes for the GRP device. It contains the address of the handler for each of the function codes, most are in fact error generators:

TO	FUNCTION
71B6H	0, open
71C2H	2, close
6E86H	4, random
7196H	6, sequential output
475AH	8, sequential input
475AH	10, loc
475AH	12, lof
475AH	14, eof
475AH	16, fpos
475AH	18, putback

```
Address... 7196H
```

This is the dispatcher sequential output routine for the GRP device. SCRMOD is first checked and an "`Illegal function call`" error generated (475AH) if the screen is in either text mode. The character to output is taken from register C and control transfers to the GRPPRT standard routine.

```
Address... 71A2H
```

This table is used by the device dispatcher when decoding function codes for the CRT device. It contains the address of the handler for each of the function codes, most are in fact error generators:

TO	FUNCTION
71B6H	0, open
71C2H	2, close
6E86H	4, random
71C3H	6, sequential output
475AH	8, sequential input
475AH	10, loc
475AH	12, lof
475AH	14, eof
475AH	16, fpos
475AH	18, putback

```
Address... 71B6H
```

This is the dispatcher open routine for the CRT, LPT and GRP devices. The required mode, in register E, is checked and a "`Bad file name`" error generated (6E6BH) for input or append. The FCB address is then placed in PTRFIL, the mode in byte 0 of the FCB and the routine terminates. Note that the Z80 RET instruction at the end of this routine (71C2H) is the dispatcher close routine for the CRT, LPT and GRP devices.

```
Address... 71C3H
```

This is the dispatcher sequential output routine for the CRT device. The character to output is taken from register C and control transfers to the CHPUT standard routine.

```
Address... 71C7H
```

This table is used by the dispatcher when decoding function codes for the CAS device. It contains the address of the handler for each of the function codes, several are error generators:

TO	FUNCTION
71DBH	0, open

TO	FUNCTION
7205H	2, close
6E86H	4, random
722AH	6, sequential output
723FH	8, sequential input
475AH	10, loc
475AH	12, lof
726DH	14, eof
475AH	16, fpos
727CH	18, putback

```
Address... 71DBH
```

This is the dispatcher open routine for the CAS device. The current I/O buffer position, held in byte 6 of the FCB, and CASPRV, which holds any putback character are both zeroed. The required mode, supplied in register E, is examined and a "`Bad file name`" error generated (6E6BH) for append or random modes. For output mode the identification block is then written to cassette (7125H) while for input mode the correct identification block is located on the cassette (70B8H). The FCB address is then placed in PTRFIL, the mode in byte 0 of the FCB and the routine terminates.

```
Address... 7205H
```

This is the dispatcher close routine for the CAS device. Byte 0 of the FCB is examined and, if the mode is input, CASPRV is zeroed and the routine terminates. Otherwise the remainder of the I/O buffer is filled with end of file characters (1AH) and the I/O buffer contents written to cassette (722FH). CASPRV is then zeroed and the routine terminates.

```
Address... 722AH
```

This is the dispatcher sequential output routine for the CAS device. The character to output is taken from register C and placed in the next free position in the I/O buffer (728BH). Byte 6 of the FCB, the I/O buffer position, is then incremented. If the I/O buffer position has wrapped round to zero this means that there are two hundred and fifty-six characters in the I/O buffer and it has to be written to cassette. The cassette motor is turned on, a short header is written to cassette (72F8H) followed by the I/O buffer contents (72DEH), and the motor is turned off via the TAPOOF standard routine.

```
Address... 723FH
```

This is the dispatcher sequential input routine for the CAS device. CASPRV is first checked (72BEH) to determine whether it contains a character which has been putback, in which case its contents will be non-zero. If so the routine terminates with the character in register A. Otherwise the I/O buffer position is checked (729BH) to determine whether it contains any characters. If the I/O buffer is empty the cassette motor is turned on and the header located (72E9H). Two hundred and fifty-six characters are then read in (72D4H), the cassette motor turned off via the TAPION standard routine and the I/O buffer position reset to zero. The character is then taken from the current I/O buffer position and the position incremented. Finally the character is checked to see if it is the end of file character (1AH). If it is not the routine terminates with the character in register A and FLAG NC. Otherwise the end of file character is placed in CASPRV, so that succeeding sequential input requests will always return the end of file condition, and the routine terminates with FLAG C.

```
Address... 726DH
```

This is the dispatcher eof routine for the CAS device. The next character is input (723FH) and placed in CASPRV. It is then tested for the end of file code (1AH) and the result placed in DAC as an integer, zero for false, FFFFH for true.

```
Address... 727CH
```

This is the dispatcher putback routine for the CAS device. The character is simply placed in CASPRV to be picked up at the next sequential input request.

```
Address... 7281H
```

This routine is used by the dispatcher close function to check if there are any characters in the I/O buffer and then zero the I/O buffer position byte in the FCB.

```
Address... 728BH
```

This routine is used by the dispatcher sequential output function to place the character in register A in the I/O buffer at the current I/O buffer position, which is then incremented.

```
Address... 729BH
```

This routine is used by the dispatcher sequential input function to collect the character at the current I/O buffer position, which is then incremented.

```
Address... 72A6H
```

This table is used by the dispatcher when decoding function codes for the LPT device. It contains the address of the handler for each of the function codes, most are in fact error generators:

TO	FUNCTION
71B6H	0, open
71C2H	2, close
6E86H	4, random
72BAH	6, sequential output
475AH	8, sequential input
475AH	10, loc
475AH	12, lof
475AH	14, eof
475AH	16, fpos
475AH	18, putback

```
Address... 72BAH
```

This is the dispatcher sequential output routine for the LPT device. The character to output is taken from register C and control transfers to the OUTDLP standard routine.

```
Address... 72BEH
```

This routine is used by the dispatcher sequential input function to check if a putback character exists in CASPRV, and if not to return Flag Z. Otherwise CASPRV is zeroed and the character tested to see if it is the end of file character (1AH). If not it returns with the character in register A and FLAG NZ,NC. Otherwise the end of file character is placed back in CASPRV and the routine returns with FLAG Z,C.

```
Address... 72CDH
```

This routine is used by various dispatcher functions to check if the mode in register E is append, if so a "`Bad file name`" error is generated (6E6BH).

```
Address... 72D4H
```

This routine is used by various dispatcher functions to read a character from the cassette. The character is read via the TAPIN standard routine and a "`Device I/O error`" generated (73B2H) if FLAG C is returned.

```
Address... 72DEH
```

This routine is used by various dispatcher functions to write a character to cassette. The character is written via the TAPOUT standard routine and a "`Device I/O error`" generated (73B2H) if FLAG C is returned.

```
Address... 72E9H
```

This routine is used by various dispatcher functions to turn the cassette motor on for input. The motor is turned on via the TAPION standard routine and a "`Device I/O error`" generated (73B2H) if FLAG C is returned.

```
Address... 72F8H
```

This routine is used by various dispatcher functions to turn the cassette motor on for output, control simply transfers to the TAPOON standard routine.

```
Address... 7304H
```

This routine is used by the Interpreter Mainloop "`OK`" point, the "`END`" statement handler and the run-clear routine to shut down the printer. PRTFLG is first zeroed and then LPTPOS tested to see if any characters have been output but left hanging in the printer's line buffer. If so a CR,LF sequence is issued to flush the printer and LPTPOS zeroed.

```
Address... 7323H
```

This routine issues a CR,LF sequence to the current output device via the OUTDO standard routine. LPTPOS or TTYPOS is then zeroed depending upon whether the printer or the screen is active.

```
Address... 7347H
```

This routine is used by the Factor Evaluator to apply the "`INKEY$`" function. The state of the keyboard buffer is examined via the CHSNS standard routine. If the buffer is empty the address of a dummy

null string descriptor is returned in DAC. Otherwise the next character is read from the keyboard buffer via the CHGET standard routine. After checking that sufficient space is available (6625H) the character is copied to the String Storage Area and the result descriptor created (6821H).

```
Address... 7367H
```

This routine is used by the "`LIST`" statement handler to output a character to the current output device via the OUTDO standard routine. If the character is a LF code then a CR code is also issued.

```
Address... 7374H
```

This routine is used by the Interpreter Mainloop to collect a line of text when input is from an I/O buffer rather than the keyboard, that is when a "`LOAD`" statement is active. Characters are sequentially input (6C71H) and placed in BUF until BUF fills up, a CR is detected or the end of file is reached. All characters are accepted apart from LF codes which are filtered out. If BUF fills up or a CR is detected the routine simply returns the line to the Mainloop. If the end of file is reached while some characters are in BUF the line is returned to the Mainloop. When end of file is reached with no characters in BUF then I/O buffer 0 is closed (6D7BH) and FILNAM checked to determine whether auto-run is required. If not control returns to the Interpreter "`OK`" point (411EH). Otherwise the system is cleared (629AH) and control transfers to the Runloop (4601H) to execute the program.

```
Address... 73B2H
```

This is the "`Device I/O error`" generator.

```
Address... 73B7H
```

This is the "`MOTOR`" statement handler. If no operand is present control transfers to the STMOTR standard routine with FFH in register A. If the "`OFF`" token (EBH) follows control transfers with 00H in register A. If the "`ON`" token (95H) follows control transfers with 01H in register A.

```
Address... 73CAH
```

This is the "`SOUND`" statement handler. The register number operand, which must be less than fourteen, is evaluated (521CH) and placed in register A. The data operand is evaluated (521CH) and bit 7 set, bit 6 reset to avoid altering the PSG auxiliary I/O port modes' The data operand is placed in register E and control transfers to the WRTPSG standard routine.

```
Address... 73E4H
```

This is a single ASCII space used by the "`PLAY`" statement handler to replace a null string operand with a one character blank string.

```
Address... 73E5H
```

This is the "`PLAY`" statement handler. The address of the "`PLAY`" command table at 752EH is placed in MCLTAB for the macro language parser and PRSCNT zeroed. The first string operand, which is obligatory, is evaluated (4C64H), its storage freed (67D0H) and its length and address placed in VCBA at bytes 2, 3 and 4. The channel's stack pointer is initialized to VCBA+33 and placed in VCBA at bytes 5 and 6' If further text is present in the statement this process is repeated for voices B and C until a maximum of three operands have been evaluated, after this a "`Syntax error`" is generated (4055H). If there are less than three string operands present an end of queue mark (FFH) is placed in the queue (7507H) of each unused voice. Register A is then zeroed, to select voice A, and control drops into the play mainloop.

```
Address... 744DH
```

This is the play mainloop. The number of free bytes in the current queue is checked (7521H) and, if less than eight bytes remain, the next voice is selected (74D6H) to avoid waiting for the queue to empty. The remaining length of the operand string is then taken from the current voice buffer and, if zero bytes remain to be parsed, the loop again skips to the next voice (74D6H). Otherwise the current string length and address are taken from the voice buffer and placed in MCLLEN and MCLPTR for the macro language parser. The old stack contents are copied from the voice buffer to the Z80 stack (6253H), MCLFLG is made non-zero and control transfers to the macro language parser (56A2H).

The macro language parser will normally scan along the string, using the "`PLAY`" statement command handlers, until the string is exhausted. However, if a music queue fills up during note generation an abnormal termination is forced back to the play mainloop (748EH) so that the next voice can be processed without waiting for the queue to empty. When control returns normally an end of queue mark is placed in the current queue (7507H) and PRSCNT is incremented to show the number of strings completed. If control returns abnormally then anything left on the Z80 stack is copied into the current voice buffer (6253H). Because of the recursive nature of the macro language parser where the "`X`" command is involved there may be a number of four byte string descriptors, marking the point where the original string was suspended, left on the Z80 stack at termination. Saving the stack contents in the voice buffer means they can be restored when the loop gets around to that voice again. Note that as there are only sixteen bytes available in each voice buffer an "`Illegal function call`" error is generated (475AH) if too much data remains on the stack. This will occur when a queue fills up and multiple, nested "X" commands exist, for example:

```
10 A$="XB$;"
20 B$="XC$;"
30 C$="XD$;"
```

```
40 D$=STRING$(150,"A")
50 PLAY A$
```

There seems to be a slight bug in this section as only fifteen bytes of stack data are allowed, instead of sixteen, before an error is generated.

When control returns from the macro language parser register A is incremented to select the next voice for processing. When all three voices have been processed INTFLG is checked and, if CTRL-STOP has been detected by the interrupt handler, control transfers to the GICINI standard routine to halt all music and terminate. Assuming bit 7 of PRSCNT shows this to be the first pass through the mainloop, that is no voice has been temporarily suspended because of a full queue, PLYCNT is incremented and interrupt dequeueing started via the STRTMS standard routine. PRSCNT is then checked to determine the number of strings completed by the macro language parser. If all three operand strings have been completed the handler terminates, otherwise control transfers back to the start of the play mainloop to try each voice again.

```
Address... 7507H
```

This routine is used by the "`PLAY`" statement handler to place an end of queue mark (FFH) in the current queue via the PUTQ standard routine. If the queue is full it waits until space becomes available.

```
Address... 7521H
```

This routine is used by the "`PLAY`" statement handler to check how much space remains in the current queue via the LFTQ standard routine. If less than eight bytes remain (the largest possible music data packet is seven bytes long) FLAG C is returned.

```
Address... 752EH
```

This table contains the valid command letters and associated addresses for the "`PLAY`" statement commands. Those commands which take a parameter, and consequently have bit 7 set in the table, are shown with an asterisk:

CMD	TO
A	763EH
B	763EH
C	763EH
D	763EH

CMD	TO
E	763EH
F	763EH
G	763EH
M*	759EH
V*	7586H
S*	75BEH
N*	7621H
O*	75EFH
R*	75FCH
T*	75E2H
L*	75C8H
X	5782H

```
Address... 755FH
```

This table is used by the "`PLAY`" statement "`A`" to "`G`" command handler to translate a note number from zero to fourteen to an offset into the tone divider table at 756EH. The note itself, rather than the note number, is shown below with each offset value:

```
16 ... A-
18 ... A
20 ... A+ or B-
22 ... B or C-
00 ... B+
00 ... C
02 ... C+ or D-
04 ... D
06 ... D+ or E-
08 ... E or F-
10 ... E+
10 ... F
12 ... F+ or G-
14 ... G
16 ... G+
```

```
Address... 756EH
```

This table contains the twelve PSG divider constants required to produce the tones of octave 1. For each constant the corresponding note and frequency are shown:

```
3421 ... C  32.698 Hz
3228 ... C+ 34.653 Hz
3047 ... D  36.712 Hz
2876 ... D+ 38.895 HZ
2715 ... E  41.201 Hz
2562 ... F  43.662 Hz
2419 ... F+ 46.243 Hz
2283 ... G  48.997 Hz
2155 ... G+ 51.908 Hz
2034 ... A  54.995 Hz
1920 ... A+ 58.261 Hz
1812 ... B  61.773 Hz
```

```
Address... 7586H
```

This is the "`PLAY`" statement "`V`" command handler. The parameter, with a default value of eight, is placed in byte 18 of the current voice buffer without altering bit 6 of the existing contents. No music data is generated.

```
Address... 759EH
```

This is the "`PLAY`" statement "`M`" command handler. The parameter, with a default value of two hundred and fifty-five, is compared with the existing modulation period contained in bytes 19 and 20 of the current voice buffer. If they are the same the routine terminates with no action. Otherwise the new modulation period is placed in the voice buffer and bit 6 set in byte 18 of the voice buffer to indicate that the new value must be incorporated into the next music data packet produced. No music data is generated.

```
Address... 75BEH
```

This is the "`PLAY`" statement "`S`" command handler. The parameter is placed in byte 18 of the current voice buffer and bit 4 of the same byte set to indicate that the new value must be incorporated into the next music data packet produced. No music data is generated. Because of the PSG characteristics the shape and volume parameters are mutually exclusive so the same byte of the voice buffers is used for both.

```
Address... 75C8H
```

This is the "`PLAY`" statement "`L`" command handler. The parameter, with a default value of four, is placed in byte 16 of the current voice buffer where it is used in the computation of succeeding note durations. No music data is generated.

```
Address... 75E2H
```

This is the "`PLAY`" statement "`T`" command handler. The parameter, with a default value of one hundred and twenty, is placed in byte 17 of the current voice buffer where it will be used in the computation of succeeding note durations. ho music data is generated.

```
Address... 75EFH
```

This is the "`PLAY`" statement "`O`" command handler. The parameter, with a default value of four, is placed in byte 15 of the current voice buffer where it is used in the computation of succeeding note frequencies. No music data is generated.

```
Address... 75FCH
```

This is the "`PLAY`" statement "`R`" command handler. The length parameter, with a default value of four, is left in register pair DE and a zero tone divider value placed in register pair HL. The existing volume value is taken from byte 18 of the current voice buffer, temporarily replaced with a zero value and control transferred to the note generator (769CH).

```
Address... 7621H
```

This is the "`PLAY`" statement "`N`" command handler. The obligatory parameter is first examined, if it is zero a rest is generated (760BH). If it is greater than ninety-six an "`Illegal function call`" error is generated (475AH). Otherwise twelve is repeatedly subtracted from the note number until underflow to obtain an octave number from one to nine in register E and a note number from zero to eleven in register C. Control then transfers to the note generator (7673H).

```
Address... 763EH
```

This is the "`PLAY`" statement "`A`" to "`G`" command handler. The note letter is first converted into a note number from zero to fourteen, this extended range being necessary because of the redundancy implicit in the notation. The table at 755FH is then used to obtain the offset into the tone divider table and the divider constant for the note placed in register pair DE. The octave value is taken from byte 15 of the current voice buffer and the divider constant halved until the correct octave is reached.

The string operand is then examined directly (56EEH) to determine whether a trailing note length parameter exists. If so it is converted (572FH) and placed in register C. If no parameter exists the default length is taken from byte 16 of the current voice buffer. The duration of the note is then computed from:

```
Duration (Interrupt ticks) = 12,000/(LENGTH*TEMPO)
```

With the normal length value (4) and tempo value (120) this gives a note duration of twenty-five interrupt ticks of 20 ms each or 0.5 seconds. The string operand is then examined (56EEH) for trailing "`.`" characters and, for each one, the duration multiplied by one and a half. Finally the resulting duration is checked and, if it is less than five interrupt ticks, it is replaced with a value of five. Thus the shortest note that can be generated on a UK machine is 0.10 seconds whatever the tempo or note length.

The music data packet, which will be three, five or seven bytes long, is then assembled in bytes 8 to 14 of the current voice buffer prior to placing it in the queue. The duration is placed in bytes 8 and 9 of the voice buffer. The volume and flag byte is taken from byte 18 and placed in byte 10 of the voice buffer with bit 7 set to indicate a volume change to the interrupt dequeuing routine. If bit 6 of the volume byte is set then the modulation period is taken from bytes 19 and 20 and added to the data packet at bytes 11 and 12. If the tone divider value is non-zero then it is added to the data packet at bytes 11 and 12 (without a modulation period) or bytes 13 and 14 (with a modulation period). Finally the byte count is mixed into the three highest bits of byte 8 of the voice buffer to complete the preparation of the music data packet.

If the tone divider value is zero, indicating a rest, the contents of SAVVOL are restored to byte 18 of the static buffer. The music data packet is then placed in the current queue via the PUTQ standard routine and the number of free bytes remaining checked (7521H). If less than eight bytes remain control transfers directly to the "`PLAY`" statement handler (748EH), otherwise control returns normally to the macro language parser.

```
Address... 7754H
```

This is the single precision constant 12,000 used in the computation of note duration.

```
Address... 7758H
```

This is the "`PUT`" statement handler. Register B is set to 80H and control drops into the "`GET`" statement handler.

```
Address... 775BH
```

This is the " `GET` " statement handler. Register B is zeroed, to distinguish " `GET` " from " `PUT` " and the next program token examined. Control then transfers to the " `PUT SPRITE` " statement handler (7AAFH) or the Disk BASIC " `GET/PUT` " statement handler (6C35H).

```
Address... 7766H
```

This is the " `LOCATE` " statement handler. If a column coordinate is present it is evaluated (521CH) and placed in register D, otherwise the current column is taken from CSRX. If a row coordinate is present it is evaluated (521CH) and placed in register E, otherwise the current row is taken from CSRY. If a cursor switch operand exists it is evaluated (521CH) and register A loaded with 78H for a zero operand (OFF) and 79H for any non-zero operand (ON). The cursor is then switched by outputting ESC, 78H/79H, " `5` " via the OUTDO standard routine. The row and column coordinates are placed in register pair HL and the cursor position set via the POSIT standard routine.

```
Address... 77A5H
```

This is the " `STOP ON/OFF/STOP` " statement handler. The address of the device's TRPTBL status byte is placed in register pair HL and control transfers to the " `ON/OFF/STOP` " routine (77CFH).

```
Address... 77ABH
```

This is the " `SPRITE ON/OFF/STOP` " statement handler. The address of the device's TRPTBL status byte is placed in register pair HL and control transfers to the " `ON/OFF/STOP` " routine (77CFH).

```
Address... 77B1H
```

This is the " `INTERVAL ON/OFF/STOP` " statement handler. As there is no specific " `INTERVAL` " token (control transfers here when an " `INT` " token is found) a check is first made on the program text for the characters " `E` " and " `R` " then the " `VAL` " token (94H). The address of the device's TRPTBL status byte is placed in register pair HL and control transfers to the " `ON/OFF/STOP` " routine (77CFH).

```
Address... 77BFH
```

This is the " `STRIG ON/OFF/STOP` " statement handler. The trigger number, from zero to four, is evaluated (7C08H) and the address of the device's TRPTBL status byte placed in register pair HL. The " `ON/OFF/STOP` " token is examined and the TRPTBL status byte modified accordingly (77FEH). Control then transfers directly to the Runloop (4612H) to avoid testing for pending interrupts until the end of the next statement.

```
Address... 77D4H
```

This is the "`KEY(n) ON/OFF/STOP`" statement handler. The key number, from one to ten, is evaluated (521CH) and the address of the devices' TRPTBL status byte placed in register pair HL. The "`ON/OFF/STOP`" token is examined and the TRPTBL status byte modified accordingly (77FEH). Bit 0 of the TRPTBL status byte, the ON bit, is then copied into the corresponding entry in FNKFLG for use during the interrupt keyscan and control transfers directly to the Runloop (4612H).

```
Address... 77FEH
```

This routine checks for the presence of one of the interrupt switching tokens and transfers control to the appropriate routine: "`ON`" (631BH), "`OFF`" (632BH) or "`STOP`" (6331H). If no token is present a "`Syntax error`" is generated (4055H).

```
Address... 7810H
```

This routine is used by the "`ON DEVICE GOSUB`" statement handler (490DH) to check the program text for a device token. Unless none of the device tokens is present, in which case Flag C is returned, the device's TRPTBL entry number is returned in register B and the maximum allowable line number operand count in register C:

DEVICE	TRPTBL#	LINE NUMBERS
KEY	00	10
STOP	10	01
SPRITE	11	01
STRIG	12	05
INTERVAL	17	01

Additionally, for "`INTERVAL`" only, the interval operand is evaluated (542FH) and placed in INTVAL and INTCNT.

```
Address... 785CH
```

This routine is used by the "`ON DEVICE GOSUB`" statement handler (490DH) to place the address of a program line in TRPTBL. The TRPTBL entry number, supplied in register B, is multiplied by three and added to the table base to point to the relevant entry. The address, supplied in register pair DE, is then placed there LSB first, MSB second.

```
Address... 786CH
```

This is the " `KEY` " statement handler. If the following character is anything other than the " `LIST` " token (93H) control transfers to the " `KEY n` " statement handler (78AEH). Each of the ten function key strings is then taken from FNKSTR and displayed via the OUTDO standard routine with a CR,LF (7328H) after each one. The DEL character (7FH) or any control character smaller than 20H is replaced with a space.

```
Address... 78AEH
```

This is the " `KEY n` ", " `KEY(n) ON/OFF/STOP` ", " `KEY ON` " and " `KEY OFF` " statement handler. If the next program text character is "(" control transfers to the " `KEY(n) ON/OFF/STOP` " statement handler (77D4H). If it is an " `ON` " token (95H) control transfers to the DSPFNK standard routine and if it is an " `OFF` " token (EBH) to the ERAFNK standard routine. Otherwise the function key number is evaluated (521CH) and the key's FNKSTR address placed in register pair DE' The string operand is evaluated (4C64H) and its storage freed (67D0H)' Up to fifteen characters are copied from the string to FNKSTR and unused positions padded with zero bytes. If a zero byte is found in the operand string an " `Illegal function call` " error is generated (475AH). Control then transfers to the FNKSB standard routine to update the function key display if it is enabled.

```
Address... 7900H
```

This routine is used by the Factor Evaluator to apply the " `TIME` " function. The contents of JIFFY are placed in DAC as a single precision number (3236H).

```
Address... 790AH
```

This routine is used by the Factor Evaluator to apply the " `CSRLIN` " function. The contents of CSRY are decremented and placed in DAC as an integer (2E9AH).

```
Address... 7911H
```

This is the " `TIME` " statement handler. The operand is evaluated (542FH) and placed in JIFFY.

```
Address... 791BH
```

This routine is used by the Factor Evaluator to apply the " `PLAY` " function. The numeric channel selection operand is evaluated (7C08H). If this is zero the contents of MUSICF are placed in DAC as an integer of value zero or FFFFH. Otherwise the channel number is used to select the appropriate bit of MUSICF and this is then converted to an integer as before.

```
Address... 7940H
```

This routine is used by the Factor Evaluator to apply the " `STICK` " function to an operand contained in DAC. The stick number is checked (521FH) and passed to the GTSTCK standard routine in register A. The result is placed in DAC as an integer (4FCFH) .

```
Address... 794CH
```

This routine is used by the Factor Evaluator to apply the " `STRIG` " function to an operand contained in DAC. The trigger number is checked (521FH) and passed to the GTTRIG standard routine in register A. The result is placed in DAC as an integer of value zero or FFFFH.

```
Address... 795AH
```

This routine is used by the Factor Evaluator to apply the " `PDL` " function to an operand contained in DAC. The paddle number is checked (521FH) and passed to the GTPDL standard routine in register A. The result is placed in DAC as an integer (4FCFH).

```
Address... 7969H
```

This routine is used by the Factor Evaluator to apply the " `PAD` " function to an operand contained in DAC. The pad number is checked (521F) and passed to the GTPAD standard routine in register A. The result is placed in DAC as an integer for pads 1, 2, 5 or 6. For pads 0, 3, 4 or 7 the result is placed in DAC as an integer of value zero or FFFFH.

```
Address... 7980H
```

This is the " `COLOR` " statement handler. If a foreground colour operand exists it is evaluated (521CH) and placed in register E, otherwise the current foreground colour is taken from FORCLR. If a background colour operand exists it is evaluated (521CH) and placed in register D, otherwise the current background colour is taken from BAKCLR. If a border colour operand exists it is evaluated (521CH) and placed in BDRCLR. The foreground colour is placed in FORCLR and ATRBYT, the background colour in BAKCLR and control transfers to the CHGCLR standard routine to modify the VDP.

```
Address... 79CCH
```

This is the " `SCREEN` " statement handler. If a mode operand exists it is evaluated (521CH) and passed to the CHGMOD standard routine in register A. If a sprite size operand exists it is evaluated (521CH) and placed in bits 0 and 1 of RG1SAV, the Workspace Area copy of VDP Mode Register 1. The VDP sprite parameters are then cleared via the CLRSPR standard routine. If a key click operand exists it is evaluated (521CH) and placed in CLIKSW, zero to disable the click and non-zero to enable it. If a baud rate operand exists it is evaluated and the baud rate set (7A2DH). If a printer mode operand exists it

is evaluated (521CH) and placed in NTMSXP, zero for an MSX printer and non- zero for a general purpose printer.

```
Address... 7A2DH
```

This routine is used to set the cassette baud rate. The operand is evaluated (521CH) and five bytes copied from CS1200 or CS2400 to LOW as appropriate.

```
Address... 7A48H
```

This is the "`SPRITE`" statement handler. If the next character is anything other than a "$" control transfers to the "`SPRITE ON/OFF/STOP`" statement handler (77ABH). SCRMOD is then checked and an "`Illegal function call`" error generated (475AH) if the screen is in 40x24 Text Mode. The sprite pattern number is evaluated and its location in the VRAM Sprite Pattern Table obtained (7AA0H). The string operand is then evaluated (4C5FH) and its storage freed (67D0H). The sprite size, obtained via the GSPSIZ standard routine, is compared with the string length and, if the string is shorter than the sprite, the Sprite Pattern Table entry is first filled with zeroes via the FILVRM standard routine. Characters are then copied from the string body to the Sprite Pattern Table via the LDIRVM standard routine until the string is exhausted or the sprite is full. If the string is longer than the sprite size any excess characters are ignored.

```
Address... 7A84H
```

This routine is used by the Factor Evaluator to apply the "`SPRITE$`" function. The sprite pattern number is evaluated and its location in the VRAM Sprite Pattern Table obtained (7A9FH). The sprite size, obtained via the GSPSIZ standard routine, is then placed in register pair BC to control the number of bytes copied. After checking that sufficient space is available in the String Storage Area (6627H) the sprite pattern is copied from VRAM via the LDIRMV standard routine and the result descriptor created (6654H). Note that as no check is made on the screen mode during this function some interesting side effects can be found, see below.

```
Address... 7A9FH
```

This routine is used by the "`SPRITE$`" statement and function to locate a sprite pattern in the VRAM Sprite Pattern Table. The pattern number operand is evaluated (7C08H) and passed to the CALPAT standard routine in register A. The pattern address is placed in register pair DE and the routine terminates.

Note that no check is made on the pattern number magnitude for differing sprite sizes. Pattern numbers up to two hundred and fifty-five are accepted even in 16x16 sprite mode when the maximum pattern number should be sixty-three. As a result VRAM addresses greater than 3FFFH will

be produced which will wrap around into low VRAM. With the " `SPRITE$` " statement this will corrupt the Character Generator Table, for example:

```
10 SCREEN 3,2
20 SPRITE$(0)=STRING$(32,255)
30 PUT SPRITE 0,(0,0), ,0
40 SPRITE$(65)=STRING$(32,255)
50 GOTO 50
```

The above puts a real sprite in the top left of the screen and then uses an illegal statement in line 40 to corrupt the VRAM just to the right of it. The " `SPRITE$` " function can also be manipulated in this way and, as there is no screen mode check, up to thirty-two bytes of the Name Table can be read in 40x24 Text Mode, for example:

```
10 SCREEN 0,2
20 PRINT"something"
30 A$=SPRITE$(64)
40 PRINT A$
```

```
Address... 7AAFH
```

This is the " `GET/PUT SPRITE` " statement handler, control is transferred here from the general " `GET/PUT` " statement handler (775BH). Register B is first checked to make sure that the statement is " `PUT` " and an " `Illegal function call` " error generated (475AH) if otherwise. SCRMOD is then checked and an " `Illegal function call` " error generated (475AH) if the screen is in 40x24 Text Mode. The sprite number operand, from zero to thirty-one, is evaluated (521CH) and passed to the CALATR standard routine to locate the four byte attribute block in the Sprite Attribute Table. If a coordinate operand exists it is evaluated and the X coordinate placed in register pair BC, the Y coordinate in register pair DE (579CH).

The Y coordinate LSB is written to byte 0 of the attribute block in VRAM via the WRTVRM standard routine. Bit 7 of the X coordinate is then examined to determine whether it is negative, that is off the left hand side of the screen. If so thirty two is added to the X coordinate and register B is set to 80H to set the early clock bit in the attribute block. For example an X coordinate of -1 (FFFFH) would be changed to +31 with an early clock. The X coordinate LSB is then written to byte 1 of the attribute block via the WRTVRM standard routine. Byte 3 of the attribute block is read in via the RDVRM standard routine, the new early clock bit is mixed in and it is then written back to VRAM via the WRTVRM standard routine.

If a colour operand is present it is evaluated (521CH), byte 3 of the attribute block is read in via the RDVRM standard routine the new colour code is mixed into the lowest four bits and it is written back to VRAM via the WRTVRM standard routine. If a pattern number operand exists it is evaluated (521CH) and checked for magnitude against the current sprite size provided by the GSPSIZ standard

routine. The maximum allowable pattern number is two hundred and fifty-five for 8x8 sprites and sixty- three for 16x16 sprites. The pattern number is written to byte 2 of the attribute block via the WRTVRM standard routine and the handler terminates.

```
Address... 7B37H
```

This is the "`VDP`" statement handler. The register number operand, from zero to seven, is evaluated (7C08H) followed by the data operand (521CH). The register number is placed in register C, the data value in register B and control transferred to the WRTVDP standard routine.

```
Address... 7B47H
```

This routine is used by the Factor Evaluator to apply the "`VDP`" function. The register number operand, from zero to eight, is evaluated (7C08H) and added to RG0SAV to locate the corresponding register image in the Workspace Area. The VDP register image is then read and placed in DAC as an integer (4FCFH).

```
Address... 7B5AH
```

This is the "`BASE`" statement handler. The VDP table number operand, from zero to nineteen, is evaluated (7C08H) followed by the base address operand (4C64H). After checking that the base address is less than 4000H (7BFEH) the VDP table number is used to locate the associated entry in the masking table at 7BA3H. The base address is ANDed with the mask and an "`Illegal function call`" error generated (475AH) if any illegal bits are set. The VDP table number is then added to TXTNAM to locate the current base address in the Workspace Area and the new base address placed there. The VDP table number is divided by five to determine which of the four screen modes the table belongs to. If this is the same as the current screen mode the new base address is also written to the VDP (7B99H).

```
Address... 7B99H
```

This routine is used by the "`BASE`" statement handler to update the VDP base addresses. The current screen mode, in register A, is examined and control transfers to the SETTXT, SETT32, SETGRP or SETMLT standard routine as appropriate. Note that this is not a full VDP initialization and that the four current table addresses (NAMBAS, CGPBAS, PATBAS and ATRBAS) which are the ones actually used by the screen routines, are not updated. This can be demonstrated with the following, where the Interpreter carries on outputting to the old VRAM Name Table:

```
10 SCREEN 0
20 BASE(0)=&H400
30 PRINT"something"
```

```
40 FOR N=1 TO 2000:NEXT
50 BASE(0)=0
```

Note also that this routine contains a bug. While SETTXT is correctly used for 40x24 Text Mode, SETGRP is used for 32x24 Text Mode and SETMLT for Graphics Mode and Multicolour Mode. Any "`BASE`" statement should therefore be immediately followed by a "`SCREEN`" statement to perform a full initialization.

```
Address... 7BA3H
```

This masking table is used by the "`BASE`" statement handler to ensure that only legal VDP base addresses are accepted. The table number and corresponding Workspace Area variable are shown with each mask:

MASK	TABLE
03FFH	00, TXTNAM
003FH	01, TXTCOL
07FFH	02, TXTCGP
007FH	03, TXTATR
07FFH	04, TXTPAT
03FFH	05, T32NAM
003FH	06, T32COL
07FFH	07, T32CGP
007FH	08, T32ATR
07FFH	09, T32PAT
03FFH	10, GRPNAM
1FFFH	11, GRPCOL
1FFFH	12, GRPCGP
007FH	13, GRPATR
07FFH	14, GRPPAT
03FFH	15, MLTNAM
003FH	16, MLTCOL
07FFH	17, MLTCGP

MASK	TABLE
007FH	18, MLTATR
07FFH	19, MLTPAT

```
Address... 7BCBH
```

This routine is used by the Factor Evaluator to apply the "`BASE`" function. The VDP table number operand, from zero to nineteen, is evaluated (7C08H) and added to TXTNAM to locate the required Workspace Area base address. This is then placed in DAC as a single precision number (3236H).

```
Address... 7BE2H
```

This is the "`VPOKE`" statement handler. The VRAM address operand is evaluated (4C64H) and checked to ensure that it is less than 4000H (7BFEH). The data operand is then evaluated (521CH) and passed to the WRTVRM standard routine in register A to write to the required address.

```
Address... 7BF5H
```

This routine is used by the Factor Evaluator to apply the "`VPEEK`" function to an operand contained in DAC. The VRAM address operand is checked to ensure it is less than 4000H (7BFEH). VRAM is then read via the RDVRM standard routine and the result placed in DAC as an integer (4FCFH).

```
Address... 7BFEH
```

This routine converts a numeric operand in DAC to an integer (2F8AH) and places it in register pair HL. If the operand is equal to or greater than 4000H, and thus outside the allowable VRAM range, an "`Illegal function call`" error is generated (475AH).

```
Address... 7C08H
```

This routine evaluates (521CH) a parenthesized numeric operand and returns it as an integer in register A. If the operand is greater than the maximum allowable value initially supplied in register A an "`Illegal function call`" error is generated (475AH).

```
Address... 7C16H
```

This is the "`DSKO$`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C1BH
```

This is the "`SET`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C20H
```

This is the "`NAME`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C25H
```

This is the "`KILL`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C2AH
```

This is the "`IPL`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C2FH
```

This is the "`COPY`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C34H
```

This is the "`CMD`" statement handler. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C39H
```

This routine is used by the Factor Evaluator to apply the "`DSKF`" function to an operand contained in DAC. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C3EH
```

This routine is used by the Factor Evaluator to apply the "`DSKI$`" function. An "`Illegal function call`" error is generated (475AH) on a standard MSX machine.

```
Address... 7C43H
```

This routine is used by the Factor Evaluator to apply the " `ATTR$` " function. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C48H
```

This is the " `LSET` " statement handler. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C4DH
```

This is the " `RSET` " statement handler. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C52H
```

This is the " `FIELD` " statement handler. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C57H
```

This routine is used by the Factor Evaluator to apply the " `MKI$` " function to an operand contained in DAC. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C5CH
```

This routine is used by the Factor Evaluator to apply the " `MKS$` " function to an operand contained in DAC. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C61H
```

This routine is used by the Factor Evaluator to apply the " `MKD$` " function to an operand contained in DAC. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C66H
```

This routine is used by the Factor Evaluator to apply the " `CVI` " function to an operand contained in DAC. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C6BH
```

This routine is used by the Factor Evaluator to apply the " `CVS` " function to an operand contained in DAC. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C70H
```

This routine is used by the Factor Evaluator to apply the " `CVD` " function to an operand contained in DAC. An " `Illegal function call` " error is generated (475AH) on a standard MSX machine.

```
Address... 7C76H
```

This routine completes the power-up initialization. At this point the entire Workspace Area is zeroed and only EXPTBL and SLTTBL have been initialized. A temporary stack is set at F376H and all one hundred and twelve hooks (560 bytes) filled with Z80 RET opcodes (C9H). HIMEM is set to F380H and the lowest RAM location found (7D5DH) and placed in BOTTOM. The one hundred and forty-four bytes of data commencing at 7F27H are copied to the Workspace Area from F380H to F40FH The function key strings are initialized via the INIFNK standard routine, ENDBUF and NLONLY are zeroed and a comma is placed in BUFMIN and a colon in KBFMIN. The address of the MSX ROM character set is taken from locations 0004H and 0005H and placed in CGPNT+1 and PRMPRV is set to point to PRMSTK. Dummy values are placed in STKTOP, MEMSIZ and VARTAB (their correct values are not known yet), one I/O buffer is allocated (7E6BH) and the Z80 SP set (62E5H). A zero byte is placed at the base of RAM, TXTTAB is set to the following location and a " `NEW` " executed (6287H).

The VDP is then initialized via the INITIO, INIT32 and CLRSPR standard routines, the cursor coordinates are set to row 11, column 10 and the sign on message " `MSX system etc.` " is displayed (6678H). After a three second delay a search is carried out for any extension ROMs (7D75H) and a further " `NEW` " executed (6287H) in case a BASIC program has been run from ROM.

Finally the identification message " `MSX BASIC etc.` " is displayed (7D29H) and control transfers to the Interpreter Mainloop " `OK` " point 411FH.

```
Address... 7D29H
```

This routine is used during power-up to enable the function key display, place the screen in 40x24 Text Mode via the INITXT standard routine, and display (6678H) the identification message " `MSX BASIC etc.` ". The amount of free memory is then computed by subtracting the contents of VARTAB from the contents of STKTOP and displayed (3412H) followed by the " `Bytes free` " message.

```
Address... 7D5DH
```

This routine is used during power-up to find the lowest RAM location. Starting at EF00H each byte is tested until one is found that cannot be written to or an address of 8000H is reached. The base address, rounded upwards to the nearest 256 byte boundary, is returned in register pair HL.

```
Address... 7D75H
```

This routine is used during power-up to perform an extension ROM search. Pages 1 and 2 (4000H to BFFFH) of each slot are examined and the results placed in SLTATR. An extension ROM has the two identification characters "`AB`" in the first two bytes to distinguish it from RAM. Information about its properties is also present in the first sixteen bytes as follows:

Reserved	Byte 10-15
BASIC Text Address MSB	Byte 9
BASIC Text Address LSB	Byte 8
DEVICE Address MSB	Byte 7
DEVICE Address LSB	Byte 6
STATEMENT Address MSB	Byte 5
STATEMENT Address LSB	Byte 4
INITIALIZE Address MSB	Byte 3
INITIALIZE Address LSB	Byte 2
42H ('B')	Byte 1
41H ('A')	Byte 0

Figure 48: ROM Header

Each page in a given slot is examined by reading the first two bytes (7E1AH) and checking for the "`AB`" characters. If a ROM is present the initialization address is read (7E1AH) and control passed to it via the CALSLT standard routine. With a games ROM there may be no return to BASIC from this point. The "`CALL`" extended statement handler address is then read (7E1AH) and bit 5 of register B set if it is valid, that is non-zero. The extended device handler address is read (7E1AH) and bit 6 of register B set if it is valid. Finally the BASIC program text address is read (7E1AH) and bit 7 of register B set if it is valid. Register B is then copied to the relevant position in SLTATR and the search continued until no more slots remain.

SLTATR is then examined for any extension ROM flagged as containing BASIC program text. If one is found its position in SLTATR is converted to a Slot ID (7E2AH) and the ROM permanently switched in via the ENASLT standard routine. VARTAB is set to C000H, as it is not known how large the Program Text Area is, TXTTAB is set to 8008H and BASROM made non-zero to disable the CTRL-STOP key. The system is cleared (629AH) and control transfers to the Runloop (4601H) to execute the BASIC program.

```
Address... 7E1AH
```

This routine is used to read two bytes from successive locations in an extension ROM. The initial address is supplied in register pair HL and the Slot ID in register C. The bytes are read via the RDSLT standard routine and returned in register pair DE. If both are zero FLAG Z is returned.

```
Address... 7E2AH
```

This routine converts the SLTATR position supplied in register B into the corresponding Slot ID in register C and ROM base address in register H. The position is first modified so that it runs from 0 to 63 rather than from 64 to 1, so that the required information is present in the form:

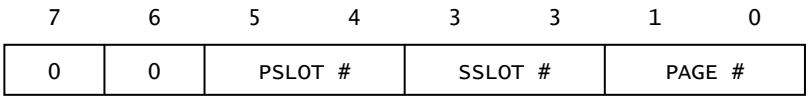


Figure 49

Bits 0 and 1 are shifted into the highest two bits of register H to form the address. Bits 4 and 5 are shifted to bits 0 and 1 of register C to form the Primary Slot number. Bits 2 and 3 are shifted to bits 2 and 3 of register C to form the Secondary Slot number and bit 7 of the corresponding EXPTBL entry copied to bit 7 of register C.

```
Address... 7E4BH
```

This is the "`MAXFILES`" statement handler. As control transfers here when a "`MAX`" token (CDH) is detected the program text is first checked for a trailing "`FILES`" token (B7H). The buffer count operand, from zero to fifteen, is then evaluated (521CH) and any existing buffers closed (6C1CH). The required number of I/O buffers are allocated (7E6BH), the system is cleared (62A7H) and control transfers directly to the Runloop (4601H).

```
Address... 7E6BH
```

This is the I/O buffer allocation routine. It is used during power-up and by the "`MAXFILES`" and "`CLEAR`" statement handlers to allocate storage for the number of I/O buffers supplied in register A. Two hundred and sixty-seven bytes are subtracted from the contents of HIMEM for every buffer to produce a new MEMSIZ value. The size of the existing String Storage Area (initially two hundred bytes) is computed by subtracting the old contents of STKTOP from the old contents of MEMSIZ, this is then subtracted from the new MEMSIZ value to produce the new STKTOP value. A further one hundred and forty bytes are subtracted for the Z80 stack and an "`Out of memory`" error generated (6275H) if this address is lower than the start of the Variable Storage Area. Otherwise the buffer count is placed in MAXFIL and MEMSIZ and STKTOP set to their new values. The caller's return address is popped, the Z80 SP set to the new position and the return address pushed back onto the stack. FILTAB is then set to the start of the I/O buffer pointer block and each pointer set to point to the

associated FCB. Finally the address of I/O buffer 0, the Interpreter's "`LOAD`" and "`SAVE`" buffer, is placed in NULBUF and the routine terminates.

```
Address... 7ED8H
```

This is the plain text message "`MSX system`" terminated by a zero byte

```
Address... 7EE4H
```

This is the plain text message "`version 1.0`" CR,LF terminated by a zero byte.

```
Address... 7EF2H
```

This is the plain text message "`MSX BASIC`" terminated by a zero byte.

```
Address... 7EFDH
```

This is the plain text message "`Copyright 1983 by Microsoft`" CR,LF terminated by a zero byte.

```
Address... 7F1BH
```

This is the plain text message "`Bytes free`" terminated by a zero byte.

```
Address... 7F27H
```

This block of one hundred and forty-four data bytes is used to initialize the Workspace Area from F380H to F40FH.

```
Address... 7FB7H
```

This seven byte patch fixes a bug in the external device parsing routine (55F8H). It checks for a zero length device name in register A and changes it to one if necessary.

```
Address... 7FBEH
```

This section of the ROM is unused and filled with zero bytes.

6. Memory Map

A maximum of 32 KB of RAM is available to the BASIC Interpreter to hold the program text, the BASIC Variables, the Z80 stack, the I/O buffers and the internal workspace. A memory map of these areas in the power-up state is shown below:

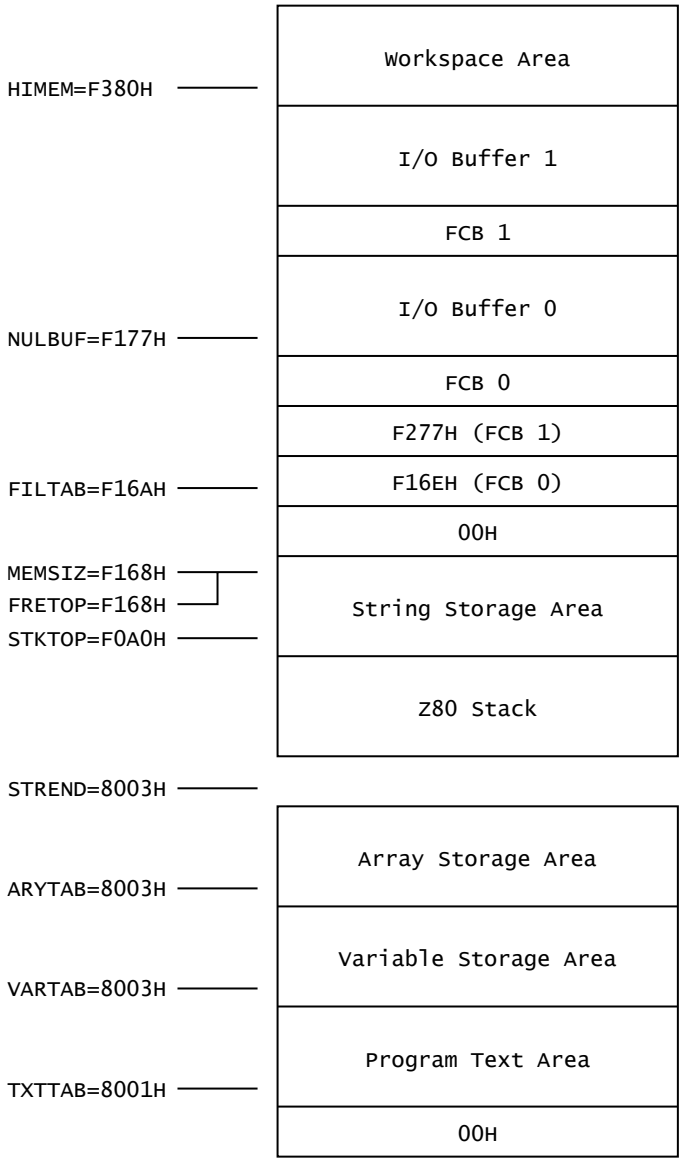


Figure 50: Memory Map 8000H to FFFFH

The Program Text Area is composed of tokenized program lines stored in line number order and terminated by a zero end link, when in the "`NEW`" state only the end link is present. The zero byte at 8000H is a dummy end of line character needed to synchronize the Runloop at the start of a program.

The Variable and Array Storage Areas are composed of string or numeric Variables and Arrays stored in the order in which they are found in the program text. Execution speed improves marginally if Variables are declared before Arrays in a program as this reduces the amount of memory to be moved upwards.

The Z80 stack is positioned immediately below the String Storage Area, the structure of the stack top is shown below:

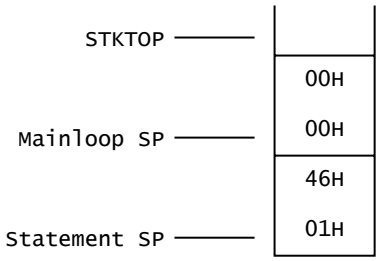


Figure 51: Z80 Stack Top

Whenever the position of the stack is altered, as a result of a " `CLEAR` " or " `MAXFILES` " statement, two zero bytes are first pushed to function as a terminator during " `FOR` " or " `GOSUB` " parameter block searches. Assuming no parameter blocks are present the Z80 SP will therefore be at STKTOP-2 within the Interpreter Mainloop and at STKTOP-4 when control transfers from the Runloop to a statement handler.

The String Storage Area is composed of the string bodies assigned to Variables or Arrays. During expression evaluation a number of intermediate string results may also be temporarily present under the permanent string heap. The zero byte following the String Storage Area is a temporary delimiter for the " `VAL` " function.

The region between the String Storage Area and HIMEM is used for I/O buffer storage. I/O buffer 0, the " `SAVE` " and " `LOAD` " buffer, is always present but the number of user buffers is determined by the " `MAXFILES` " statement. Each I/O buffer consists of a 9 byte FCB, whose address is contained in the table under FCB 0, followed by a 256 byte data buffer. The FCB contains the status of the I/O buffer as below:

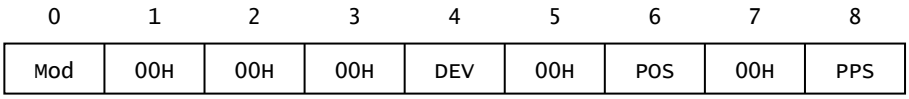


Figure 52: File Control Block

The MOD byte holds the buffer mode, the DEV byte the device code, the POS byte the current position in the buffer (0 to 255) and the PPS byte the " `PRINT` " position. The remainder of the FCB is unused on a standard MSX machine.

Workspace Area

The section of the Workspace Area from F380H to FD99H holds the BIOS/Interpreter variables. These are listed on the following pages in standard assembly language form:

```
F380H RDPRIM: OUT (0A8H),A ; Set new Primary Slot
F382H         LD E,(HL)    ; Read memory
F383H         JR WRPRM1    ; Restore old Primary Slot
```

This routine is used by the RDSLT standard routine to switch Primary Slots and read a byte from memory. The new Primary Slot Register setting is supplied in register A, the old setting in register D and the byte read returned in register E.

```
F385H WRPRIM: OUT (0A8H),A ; Set new Primary Slot
F387H         LD (HL),E    ; Write to memory
F388H WRPRM1: LD A,D       ; Get old setting
F389H         OUT (0A8H),A ; Restore old Primary Slot
F38BH         RET
```

This routine is used by the WRSLT standard routine to switch Primary Slots and write a byte to memory. The new Primary Slot Register setting is supplied in register A, the old setting in register D and the byte to write in register E.

```
F38CH CLPRIM: OUT (0A8H),A ; Set new Primary Slot
F38EH         EX AF,AF'    ; Swap to AF for call
F38FH         CALL CLPRM1  ; Do it
F392H         EX AF,AF'    ; Swap to AF
F393H         POP AF       ; Get old setting
F394H         OUT (0A8H),A ; Restore old Primary Slot
F396H         EX AF,AF'    ; Swap to AF
F397H         RET
F398H CLPRM1: JP (IX)
```

This routine is used by the CALSLT standard routine to switch Primary Slots and call an address. The new Primary Slot Register setting is supplied in register A, the old setting on the Z80 stack and the address to call in register pair IX.

```
F39AH USRTAB: DEFW 475AH   ; USR 0
F39CH         DEFW 475AH   ; USR 1
F39EH         DEFW 475AH   ; USR 2
F3A0H         DEFW 475AH   ; USR 3
F3A2H         DEFW 475AH   ; USR 4
F3A4H         DEFW 475AH   ; USR 5
F3A6H         DEFW 475AH   ; USR 6
F3A8H         DEFW 475AH   ; USR 7
F3AAH         DEFW 475AH   ; USR 8
F3ACH         DEFW 475AH   ; USR 9
```

These ten variables contain the "`USR`" function addresses. Their values are set to the Interpreter "`Illegal function call`" error generator at power-up and thereafter only altered by the "`DEFUSR`"

statement.

```
F3AEH LINL40: DEFB 37
```

This variable contains the 40x24 Text Mode screen width. Its value is set at power-up and thereafter only altered by the "`WIDTH`" statement.

```
F3AFH LINL32: DEFB 29
```

This variable contains the 32x24 Text Mode screen width. Its value is set at power-up and thereafter only altered by the "`WIDTH`" statement.

```
F3B0H LINLEN: DEFB 37
```

This variable contains the current text mode screen width. Its value is set from LINL40 or LINL32 whenever the VDP is initialized to a text mode via the INITXT or INIT32 standard routines.

```
F3B1H CRTCNT: DEFB 24
```

This variable contains the number of rows on the screen. Its value is set at power-up and thereafter unaltered.

```
F3B2H CLMLST: DEFB 14
```

This variable contains the minimum number of columns that must still be available on a line for a data item to be "`PRINT`"ed, if less space is available a CR,LF is issued first. Its value is set at power-up and thereafter only altered by the "`WIDTH`" and "`SCREEN`" statements.

```
F3B3H TXTNAM: DEFW 0000H   ; Name Table Base
F3B5H TXTCOL: DEFW 0000H   ; Colour Table Base
F3B7H TXTCGP: DEFW 0800H   ; Character Pattern Base
F3B9H TXTATR: DEFW 0000H   ; Sprite Attribute Base
F3BBH TXTPAT: DEFW 0000H   ; Sprite Pattern Base
```

These five variables contain the 40x24 Text Mode VDP base addresses. Their values are set at power-up and thereafter only altered by the "`BASE`" statement.

```
F3BDH T32NAM: DEFW 1800H   ; Name Table Base
F3BFH T32COL: DEFW 2000H   ; Colour Table Base
F3C1H T32CGP: DEFW 0000H   ; Character Pattern Base
```

```
F3C3H T32ATR: DEFW 1B00H   ; Sprite Attribute Base
F3C5H T32PAT: DEFW 3800H   ; Sprite Pattern Base
```

These five variables contain the 32x24 Text Mode VDP base addresses. Their values are set at power-up and thereafter only altered by the " `BASE` " statement.

```
F3C7H GRPNAM: DEFW 1800H   ; Name Table Base
F3C9H GRPCOL: DEFW 2000H   ; Colour Table Base
F3CBH GRPCGP: DEFW 0000H   ; Character Pattern Base
F3CDH GRPATR: DEFW 1B00H   ; Sprite Attribute Base
F3CFH GRPPAT: DEFW 3800H   ; Sprite Pattern Base
```

These five variables contain the Graphics Mode VDP base addresses. Their values are set at power-up and thereafter only altered by the " `BASE` " statement.

```
F3D1H MLTNAM: DEFW 0800H   ; Name Table Base
F3D3H MLTCOL: DEFW 0000H   ; Colour Table Base
F3D5H MLTCGP: DEFW 0000H   ; Character Pattern Base
F3D7H MLTATR: DEFW 1B00H   ; Sprite Attribute Base
F3D9H MLTPAT: DEFW 3800H   ; Sprite Pattern Base
```

These five variables contain the Multicolour Mode VDP base addresses. Their values are set at power-up and thereafter only altered by the " `BASE` " statement.

```
F3DBH CLIKSW: DEFB 01H
```

This variable controls the interrupt handler key click: 00H=Off, NZ=On. Its value is set at power-up and thereafter only altered by the " `SCREEN` " statement.

```
F3DCH CSRY:   DEFB 01H
```

This variable contains the row coordinate (from 1 to CTRCNT) of the text mode cursor.

```
F3DDH CSRX:   DEFB 01H
```

This variable contains the column coordinate (from 1 to LINLEN) of the text mode cursor. Note that the BIOS cursor coordinates for the home position are 1,1 whatever the screen width.

```
F3DEH CNSDFG: DEFB FFH
```

This variable contains the current state of the function key display: 00H=Off, NZ=On.

```
F3DFH RG0SAV: DEFB 00H
F3E0H RG1SAV: DEFB F0H
F3E1H RG2SAV: DEFB 00H
F3E2H RG3SAV: DEFB 00H
F3E3H RG4SAV: DEFB 01H
F3E4H RG5SAV: DEFB 00H
F3E5H RG6SAV: DEFB 00H
F3E6H RG7SAV: DEFB F4H
```

These eight variables mimic the state of the eight write-only VDP Mode Registers. The values shown are for 40x24 Text Mode.

```
F3E7H STATFL: DEFB CAH
```

This variable is continuously updated by the interrupt handler with the contents of the VDP Status Register.

```
F3E8H TRGFLG: DEFB F1H
```

This variable is continuously updated by the interrupt handler with the state of the four joystick trigger inputs and the space key.

```
F3E9H FORCLR: DEFB 0FH     ; White
```

This variable contains the current foreground colour. Its value is set at power-up and thereafter only altered by the "`COLOR`" statement. The foreground colour is used by the CLRSPR standard routine to set the sprite colour and by the CHGCLR standard routine to set the 1 pixel colour in the text modes. It also functions as the graphics ink colour as it is copied to ATRBYT by the GRPPRT standard routine and used throughout the Interpreter as the default value for any optional colour operand.

```
F3EAH BAKCLR: DEFB 04H     ; Dark blue
```

This variable contains the current background colour. Its value is set at power-up and thereafter only altered by the "`COLOR`" statement. The background colour is used by the CLS standard routine to clear the screen in the graphics modes and by the CHGCLR standard routine to set the 0 pixel colour in the text modes.

```
F3EBH BDRCLR: DEFB 04H     ; Dark blue
```

This variable contains the current border colour. Its value is set at power-up and thereafter only altered by the "`COLOR`" statement. The border colour is used by the CHGCLR standard routine in

32x24 Text Mode, Graphics Mode and Multicolour Mode to set the border colour.

```
F3ECH MAXUPD: DEFB C3H
F3EDH         DEFW 0000H
```

These two bytes are filled in by the "`LINE`" statement handler to form a Z80 JP to the RIGHTC, LEFTC, UPC or DOWNC standard routines.

```
F3EFH MINUPD: DEFB C3H
F3F0H         DEFW 0000H
```

These two bytes are filled in by the "`LINE`" statement handler to form a Z80 JP to the RIGHTC, LEFTC, UPC or DOWNC standard routines.

```
F3F2H ATRBYT: DEFB 0FH
```

This variable contains the graphics ink colour used by the SETC and NSETCX standard routines.

```
F3F3H QUEUES: DEFW F959H
```

This variable contains the address of the control blocks for the three music queues. Its value is set at power-up and thereafter unaltered.

```
F3F5H FRCNEW: DEFB FFH
```

This variable contains a flag to distinguish the two statements in the "`CLOAD/CLOAD?`" statement handler: 00H=CLOAD, FFH=CLOAD?.

```
F3F6H SCNCNT: DEFB 01H
```

This variable is used as a counter by the interrupt handler to control the rate at which keyboard scans are performed.

```
F3F7H REPCNT: DEFB 01H
```

This variable is used as a counter by the interrupt handler to control the key repeat rate.

```
F3F8H PUTPNT: DEFW FBF0H
```

This variable contains the address of the put position in KEYBUF.

```
F3FAH GETPNT: DEFW FBF0H
```

This variable contains the address of the get position in KEYBUF.

```
F3FCH CS1200: DEFB 53H     ; LO cycle 1st half
F3FDH         DEFB 5CH     ; LO cycle 2nd half
F3FEH         DEFB 26H     ; HI cycle 1st half
F3FFH         DEFB 2DH     ; HI cycle 2nd half
F400H         DEFB 0FH     ; Header cycle count
```

These five variables contain the 1200 baud cassette parameters. Their values are set at power-up and thereafter unaltered.

```
F401H CS2400: DEFB 25H     ; LO cycle 1st half
F402H         DEFB 2DH     ; LO cycle 2nd half
F403H         DEFB 0EH     ; HI cycle 1st half
F404H         DEFB 16H     ; HI cycle 2nd half
F405H         DEFB 1FH     ; Header cycle count
```

These five variables contain the 2400 baud cassette parameters. Their values are set at power-up and thereafter unaltered.

```
F406H LOW:    DEFB 53H     ; LO cycle 1st half
F407H         DEFB 5CH     ; LO cycle 2nd half
F408H HIGH:   DEFB 26H     ; HI cycle 1st half
F409H         DEFB 2DH     ; HI cycle 2nd half
F40AH HEADER: DEFB 0FH     ; Header cycle count
```

These five variables contain the current cassette parameters. Their values are set to 1200 baud at power-up and thereafter only altered by the "`CSAVE`" and "`SCREEN`" statements.

```
F40BH ASPCT1: DEFW 0100H
```

This variable contains the reciprocal of the default "`CIRCLE`" aspect ratio multiplied by 256. Its value is set at power-up and thereafter unaltered.

```
F40DH ASPCT2: DEFW 01C0H
```

This variable contains the default "`CIRCLE`" aspect ratio multiplied by 256. Its value is set at power-up and thereafter unaltered. The aspect ratio is present in two forms so that the "`CIRCLE`" statement handler can select the appropriate one immediately rather than needing to examine and possibly reciprocate it as is the case with an operand in the program text.

```
F40FH ENDPRG: DEFB ":"
F410H         DEFB 00H
F411H         DEFB 00H
FE12H         DEFB 00H
F413H         DEFB 00H
```

These five bytes form a dummy program line. Their values are set at power-up and thereafter unaltered. The line exists in case an error occurs in the Interpreter Mainloop before any tokenized text is available in KBUF. If an "`ON ERROR GOTO`" is active at this time then it provides some text for the "`RESUME`" statement to terminate on.

```
F414H ERRFLG: DEFB 00H
```

This variable is used by the Interpreter error handler to save the error number.

```
F415H LPTPOS: DEFB 00H
```

This variable is used by the "`LPRINT`" statement handler to hold the current position of the printer head.

```
F416H PRTFLG: DEFB 00H
```

This variable determines whether the OUTDO standard routine directs its output to the screen or to the printer: 00H=Screen, 01H=Printer.

```
F417H NTMSXP: DEFB 00H
```

This variable determines whether the OUTDO standard routine will replace headered graphics characters directed to the printer with spaces: 00H=Graphics, NZ=Spaces. Its value is set at power-up and thereafter only altered by the "`SCREEN`" statement.

```
F418H RAWPRT: DEFB 00H
```

This variable determines whether the OUTDO standard routine will modify control and headered graphics characters directed to the printer: 00H=Modify, NZ=Raw. Its value is set at power-up and thereafter unaltered.

```
F419H VLZADR: DEFW 0000H
F41BH VLZDAT: DEFB 00H
```

These variables contain the address and value of any character temporarily removed by the "`VAL`" function.

```
F41CH CURLIN: DEFW FFFFH
```

This variable contains the current Interpreter line number. A value of FFFFH denotes direct mode.

```
F41EH KBFMIN: DEFB ":"
```

This byte provides a dummy prefix to the tokenized text contained in KBUF. Its function is similar to that of ENDPRG but is used for the situation where an error occurs within a direct statement.

```
F41FH KBUF:   DEFS 318
```

This buffer contains the tokenized form of the input line collected by the Interpreter Mainloop. When a direct statement is executed the contents of this buffer form the program text.

```
F55DH BUFMIN: DEFB ","
```

This byte provides a dummy prefix to the text contained in BUF. It is used to synchronize the "`INPUT`" statement handler as it starts to analyze the input text.

```
F55EH BUF:    DEFS 259
```

This buffer contains the text collected from the console by the INLIN standard routine.

```
F661H TTYPOS: DEFB 00H
```

This variable is used by the "`PRINT`" statement handler to hold the current screen position (Teletype!).

```
F662H DIMFLG: DEFB 00H
```

This variable is normally zero but is set by the "`DIM`" statement handler to control the operation of the variable search routine.

```
F663H VALTYP: DEFB 02H
```

This variable contains the type code of the operand currently contained in DAC: integer, 3=String, 4=Single Precision, 8=Double Precision.

```
F664H DORES:  DEFB 00H
```

This variable is normally zero but is set to prevent the tokenization of unquoted keywords following a "`DATA`" token.

```
F665H DONUM:  DEFB 00H
```

This variable is normally zero but is set when a numeric constant follows one of the keywords `GOTO`, `GOSUB`, `THEN`, etc., and must be tokenized to the special line number operand form.

```
F666H CONTXT: DEFW 0000H
```

This variable is used by the CHRGTR standard routine to save the address of the character following a numeric constant in the program text.

```
F668H CONSAV: DEFB 00H
```

This variable is used by the CHRGTR standard routine to save the token of a numeric constant found in the program text.

```
F669H CONTYP: DEFB 00H
```

This variable is used by the CHRGTR standard routine to save the type of a numeric constant found in the program text.

```
F66AH CONLO:  DEFS 8
```

This buffer is used by the CHRGTR standard routine to save the value of a numeric constant found in the program text.

```
F672H MEMSIZ: DEFW F168H
```

This variable contains the address of the top of the String Storage Area. Its value is set at power-up and thereafter only altered by the "`CLEAR`" and "`MAXFILES`" statements.

```
F674H STKTOP: DEFW F0A0H
```

This variable contains the address of the top of the Z80 stack. Its value is set at power-up to MEMSIZ-200 and thereafter only altered by the "`CLEAR`" and "`MAXFILES`" statements.

```
F676H TXTTAB: DEFW 8001H
```

This variable contains the address of the first byte of the Program Text Area. Its value is set at power-up and thereafter unaltered.

```
F678H TEMPPT: DEFW F67AH
```

This variable contains the address of the next free location in TEMPST.

```
F67AH TEMPST: DEFS 30
```

This buffer is used to store string descriptors. It functions as a stack with string producers pushing their results and string consumers popping them.

```
F698H DSCTMP: DEFS 3
```

This buffer is used by the string functions to hold a result descriptor while it is being constructed.

```
F69BH FRETOP: DEFW F168H
```

This variable contains the address of the next free location in the String Storage Area. When the area is empty FRETOP is equal to MEMSIZ.

```
F69DH TEMP3: DEFW 0000H
```

This variable is used for temporary storage by various parts of the Interpreter.

```
F69FH TEMP8:  DEFW 0000H
```

This variable is used for temporary storage by various parts of the Interpreter.

```
F6A1H ENDFOR: DEFW 0000H
```

This variable is used by the "`FOR`" statement handler to hold the end of statement address during construction of a parameter block.

```
F6A3H DATLIN: DEFW 0000H
```

This variable contains the line number of the current "`DATA`" item in the program text.

```
F6A5H SUBFLG: DEFB 00H
```

This variable is normally zero but is set by the "`ERASE`", "`FOR`", "`FN`" and "`DEF FN`" handlers to control the processing of subscripts by the variable search routine.

```
F6A6H FLGINP: DEFB 00H
```

This variable contains a flag to distinguish the two statements in the "`READ/INPUT`" statement handler: 00H=INPUT, NZ=READ.

```
F6A7H TEMP:   DEFW 0000H
```

This variable is used for temporary storage by various parts of the Interpreter.

```
F6A9H PTRFLG: DEFB 00H
```

This variable is normally zero but is set if any line number operands in the Program Text Area have been converted to pointers.

```
F6AAH AUTFLG: DEFB 00H
```

This variable is normally zero but is set when "`AUTO`" mode is turned on.

```
F6ABH AUTLIN: DEFW 0000H
```

This variable contains the current "`AUTO`" line number.

```
F6ADH AUTINC: DEFW 0000H
```

This variable contains the current "`AUTO`" line number increment.

```
F6AFH SAVTXT: DEFW 0000H
```

This variable is updated by the Runloop at the start of every statement with the current location in the program text. It is used during error recovery to set ERRTXT for the "`RESUME`" statement handler and OLDTXT for the "`CONT`" statement handler.

```
F6B1H SAVSTK: DEFW F09EH
```

This variable is updated by the Runloop at the start of every statement with the current Z80 SP for error recovery purposes.

```
F6B3H ERRLIN: DEFW 0000H
```

This variable is used by the error handler to hold the line number of the program line generating an error.

```
F6B5H DOT:    DEFW 0000H
```

This variable is updated by the Mainloop and the error handler with the current line number for use with the "." parameter.

```
F6B7H ERRTXT: DEFW 0000H
```

This variable is updated from SAVTXT by the error handler for use by the "`RESUME`" statement handler.

```
F6B9H ONELIN: DEFW 0000H
```

This variable is set by the "`ON ERROR GOTO`" statement handler with the address of the program line to execute when an error occurs.

```
F6BBH ONEFLG: DEFB 00H
```

This variable is normally zero but is set by the error handler when control transfers to an "`ON ERROR GOTO`" statement. This is to prevent a loop developing if an error occurs inside the error recovery statements.

```
F6BCH TEMP2:  DEFW 0000H
```

This variable is used for temporary storage by various parts of the Interpreter.

```
F6BEH OLDLIN: DEFW 0000H
```

This variable contains the line number of the terminating program line. It is set by the "`END`" and "`STOP`" statement handlers for use with the "`CONT`" statement.

```
F6C0H OLDTXT: DEFW 0000H
```

This variable contains the address of the terminating program statement.

```
F6C2H VARTAB: DEFW 8003H
```

This variable contains the address of the first byte of the Variable Storage Area.

```
F6C4H ARYTAB: DEFW 8003H
```

This variable contains the address of the first byte of the Array Storage Area.

```
F6C6H STREND: DEFW 8003H
```

This variable contains the address of the byte following the Array Storage Area.

```
F6C8H DATPTR: DEFW 8000H
```

This variable contains the address of the current "`DATA`" item in the program text.

```
F6CAH DEFTBL: DEFB 08H     ; A
F6CBH         DEFB 08H     ; B
F6CCH         DEFB 08H     ; C
F6CDH         DEFB 08H     ; D
F6CEH         DEFB 08H     ; E
F6CFH         DEFB 08H     ; F
F6D0H         DEFB 08H     ; G
F6D1H         DEFB 08H     ; H
F6D2H         DEFB 08H     ; I
F6D3H         DEFB 08H     ; J
F6D4H         DEFB 08H     ; K
F6D5H         DEFB 08H     ; L
F6D6H         DEFB 08H     ; M
F6D7H         DEFB 08H     ; N
F6D8H         DEFB 08H     ; O
F6D9H         DEFB 08H     ; P
F6DAH         DEFB 08H     ; Q
F6DBH         DEFB 08H     ; R
F6DCH         DEFB 08H     ; S
F6DDH         DEFB 08H     ; T
F6DEH         DEFB 08H     ; U
F6DFH         DEFB 08H     ; V
F6E0H         DEFB 08H     ; W
F6E1H         DEFB 08H     ; X
```

```
F6E2H         DEFB 08H     ; Y
F6E3H         DEFB 08H     ; Z
```

These twenty-six variables contain the default type for each group of BASIC Variables. Their values are set to double precision at power-up, "`NEW`" and "`CLEAR`" and thereafter altered only by the "`DEF`" group of statements.

```
F6E4H PRMSTK: DEFW 0000H
```

This variable contains the base address of the previous "`FN`" parameter block on the Z80 stack. It is used during string garbage collection to travel from block to block on the stack.

```
F6E6H PRMLEN: DEFW 0000H
```

This variable contains the length of the current "`FN`" parameter block in PARM1.

```
F6E8H PARM1 : DEFS 100
```

This buffer contains the local Variables belonging to the "`FN`" function currently being evaluated.

```
F74CH PRMPRV: DEFW F6E4H
```

This variable contains the address of the previous "`FN`" parameter block. It is actually a constant used to ensure that string garbage collection commences with the current parameter block before proceeding to those on the stack.

```
F74EH PRMLN2: DEFW 0000H
```

This variable contains the length of the "`FN`" parameter block being constructed in PARM2

```
F750H PARM2:  DEFS 100
```

This buffer is used to construct the local Variables owned by the current "`FN`" function.

```
F7B4H PRMFLG: DEFB 00H
```

This variable is used during a Variable search to indicate whether local or global Variables are being examined.

```
F7B5H ARYTA2: DEFW 0000H
```

This variable is used during a Variable search to hold the termination address of the storage area being examined.

```
F7B7H NOFUNS: DEFB 00H
```

This variable is normally zero but is set by the "`FN`" function handler to indicate to the variable search routine that local Variables are present.

```
F7B8H TEMP9:  DEFW 0000H
```

This variable is used for temporary storage by various parts of the Interpreter.

```
F7BAH FUNACT: DEFW 0000H
```

This variable contains the number of currently active "`FN`" functions.

```
F7BCH SWPTMP: DEFS 8
```

This buffer is used to hold the first operand in a "`SWAP`" statement.

```
F7C4H TRCFLG: DEFB 00H
```

This variable is normally zero but is set by the "`TRON`" statement handler to turn on the trace facility.

```
F7C5H FBUFFR: DEFS 43
```

This buffer is used to hold the text produced during numeric output conversion.

```
F7F0H DECTMP: DEFW 0000H
```

This variable is used for temporary storage by the double precision division routine.

```
F7F2H DECTM2: DEFW 0000H
```

This variable is used for temporary storage by the double precision division routine.

```
F7F4H DECCNT: DEFB 00H
```

This variable is used by the double precision division routine to hold the number of non-zero bytes in the mantissa of the second operand.

```
F7F6H DAC:    DEFS 16
```

This buffer functions as the Interpreter's primary accumulator during expression evaluation.

```
F806H HOLD8:  DEFS 65
```

This buffer is used by the double precision multiplication routine to hold the multiples of the first operand.

```
F847H ARG:    DEFS 16
```

This buffer functions as the Interpreter's secondary accumulator during expression evaluation.

```
F857H RNDX:   DEFS 8
```

This buffer contains the current double precision random number.

```
F85FH MAXFIL: DEFB 01H
```

This variable contains the number of currently allocated user I/O buffers. Its value is set to 1 at power-up and thereafter only altered by the "`MAXFILES`" statement.

```
F860H FILTAB: DEFW F16AH
```

This variable contains the address of the pointer table for the I/O buffer FCBs.

```
F862H NULBUF: DEFW F177H
```

This variable contains the address of the first byte of the data buffer belonging to I/O buffer 0.

```
F864H PTRFIL: DEFW 0000H
```

This variable contains the address of the currently active I/O buffer FCB.

```
F866H FILNAM: DEFS 11
```

This buffer holds a user-specified filename. It is eleven characters long to allow for disc file specs such as "`FILENAME.BAS`".

```
F871H FILNM2: DEFS 11
```

This buffer holds a filename read from an I/O device for comparison with the contents of FILNAM.

```
F87CH NLONLY: DEFB 00H
```

This variable is normally zero but is set during a program "`LOAD`". Bit 0 is used to prevent I/O buffer 0 being closed during loading and bit 7 to prevent the user I/O buffers being closed if auto-run is required.

```
F87DH SAVEND: DEFW 0000H
```

This variable is used by the "`BSAVE`" statement handler to hold the end address of the memory block to be saved.

```
F87FH FNKSTR: DEFS 160
```

This buffer contains the ten sixteen-character function key strings. Their values are set at power-up and thereafter only altered by the "`KEY`" statement.

```
F91FH CGPNT:  DEFB 00H     ; Slot ID
F920H         DEFW 1BBFH   ; Address
```

These variables contain the location of the character set copied to the VDP by the INITXT and INIT32 standard routines. Their values are set to the MSX ROM character set at power-up and thereafter unaltered.

```
F922H NAMBAS: DEFW 0000H
```

This variable contains the current text mode VDP Name Table base address. Its value is set from TXTNAM or T32NAM whenever the VDP is initialized to a text mode via the INITXT or INIT32 standard routines.

```
F924H CGPBAS: DEFW 0800H
```

This variable contains the current text mode VDP Character Pattern Table base address. Its value is set from TXTCGP or T32CGP whenever the VDP is initialized to a text mode via the INITXT or INIT32 standard routines.

```
F926H PATBAS: DEFW 3800H
```

This variable contains the current VDP Sprite Pattern Table base address. Its value is set from T32PAT, GRPPAT or MLTPAT whenever the VDP is initialized via the INIT32, INIGRP or INIMLT standard routines.

```
F928H ATRBAS: DEFW 1B00H
```

This variable contains the current VDP Sprite Attribute Table base address. Its value is set from T32ATR, GRPATR or MLTATR whenever the VDP is initialized via the INIT32, INIGRP or INIMLT standard routines.

```
F92AH CLOC:   DEFW 0000H   ; Pixel location
F92CH CMASK:  DEFB 80H     ; Pixel Mask
```

These variables contain the current pixel physical address used by the RIGHTC, LEFTC, UPC, TUPC, DOWNC, TDOWNC, FETCHC, STOREC, READC, SETC, NSETCX, SCANR and SCANL standard routines. CLOC holds the address of the byte containing the current pixel and CMASK defines the pixel within that byte.

```
F92DH MINDEL: DEFW 0000H
```

This variable is used by the "`LINE`" statement handler to hold the minimum difference between the end points of the line.

```
F92FH MAXDEL: DEFW 0000H
```

This variable is used by the "`LINE`" statement handler to hold the maximum difference between the end points of the line.

```
F931H ASPECT: DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler to hold the current aspect ratio. This is stored as a single byte binary fraction so an aspect ratio of 0.75 would become 00C0H. The MSB is only required if the aspect ratio is exactly 1.00, that is 0100H.

```
F933H CENCNT: DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler to hold the point count of the end angle.

```
F935H CLINEF: DEFB 00H
```

This variable is used by the "`CIRCLE`" statement handler to hold the two line flags. Bit 0 is set if a line is required from the start angle to the centre and bit 7 set if one is required from the end angle.

```
F936H CNPNTS: DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler to hold the number of points within a forty-five degree segment.

```
F938H CPLOTF: DEFB 00H
```

This variable is normally zero but is set by the "`CIRCLE`" statement handler if the end angle is smaller than the start angle. It is used to determine whether the pixels should be set "inside" the angles or "outside" them.

```
F939H CPCNT:  DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler to hold the point count within the current forty-five degree segment, this is in fact the Y coordinate.

```
F93BH CPCNT8: DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler to hold the total point count of the present position.

```
F93DH CRCSUM: DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler as the point computation counter.

```
F93FH CSTCNT: DEFW 0000H
```

This variable is used by the "`CIRCLE`" statement handler to hold the point count of the start angle.

```
F941H CSCLXY: DEFB 00H
```

This variable is used by the "`CIRCLE`" statement handler as a flag to determine in which direction the elliptic squash is to be applied: 00H=Y, 01H=X.

```
F942H CSAVEA: DEFW 0000H
```

This variable is used for temporary storage by the SCANR standard routine.

```
F944H CSAVEM: DEFB 00h
```

This variable is used for temporary storage by the SCANR standard routine.

```
F945H CXOFF:  DEFW 0000H
```

This variable is used for temporary storage by the "`CIRCLE`" statement handler.

```
F947H CYOFF:  DEFW 0000H
```

This variable is used for temporary storage by the "`CIRCLE`" statement handler.

```
F949H LOHMSK: DEFB 00H
```

This variable is used by the "`PAINT`" statement handler to hold the leftmost position of a LH excursion.

```
F94AH LOHDIR: DEFB 00H
```

This variable is used by the "`PAINT`" statement handler to hold the new paint direction required by a LH excursion.

```
F94BH LOHADR: DEFW 0000H
```

This variable is used by the "`PAINT`" statement handler to hold the leftmost position of a LH excursion.

```
F94DH LOHCNT: DEFW 0000H
```

This variable is used by the "`PAINT`" statement handler to hold the size of a LH excursion.

```
F94FH SKPCNT: DEFW 0000H
```

This variable is used by the "`PAINT`" statement handler to hold the skip count returned by the SCANR standard routine.

```
F951H MOVCNT: DEFW 0000H
```

This variable is used by the "`PAINT`" statement handler to hold the movement count returned by the SCANR standard routine.

```
F953H PDIREC: DEFB 00H
```

This variable is used by the "`PAINT`" statement handler to hold the current paint direction: 40H=Down, C0H=Up, 00H=Terminate.

```
F954H LFPROG: DEFB 00H
```

This variable is normally zero but is set by the "`PAINT`" statement handler if there has been any leftwards progress.

```
F955H RTPROG: DEFB 00H
```

This variable is normally zero but is set by the "`PAINT`" statement handler if there has been any rightwards progress.

```
F956H MCLTAB: DEFW 0000H
```

This variable contains the address of the command table to be used by the macro language parser. The "`DRAW`" table is at 5D83H and the "`PLAY`" table at 752EH.

```
F958H MCLFLG: DEFB 00H
```

This variable is zero if the macro language parser is being used by the "`DRAW`", statement handler and non-zero if it is being used by "`PLAY`".

```
F959H QUETAB: DEFB 00H     ; AQ Put position
F95AH         DEFB 00H     ; AQ Get position
F95BH         DEFB 00H     ; AQ Putback flag
```

```
F95CH         DEFB 7FH     ; AQ Size
F95DH         DEFW F975H   ; AQ Address

F95FH         DEFB 00H     ; BQ Put position
F960H         DEFB 00H     ; BQ Get position
F961H         DEFB 00H     ; BQ Putback flag
F962H         DEFB 7FH     ; BQ Size
F963H         DEFW F9F5H   ; BQ Address

F965H         DEFB 00H     ; CQ Put position
F966H         DEFB 00H     ; CQ Get position
F967H         DEFB 00H     ; CQ Putback flag
F968H         DEFB 7FH     ; CQ Size
F969H         DEFW FA75H   ; CQ Address

F96BH         DEFB 00H     ; RQ Put position
F96CH         DEFB 00H     ; RQ Get position
F96DH         DEFB 00H     ; RQ Putback flag
F96EH         DEFB 00H     ; RQ Size
F96FH         DEFW 0000H   ; RQ Address
```

These twenty-four variables form the control blocks for the three music queues (VOICAQ, VOICBQ and VOICCQ) and the RS232 queue. The three music control blocks are initialized by the GICINI standard routine and thereafter maintained by the interrupt handler and the PUTQ standard routine. The RS232 control block is unused in the current MSX ROM.

```
F971H QUEBAK: DEFB 00H     ; AQ Putback character
F972H         DEFB 00H     ; BQ Putback character
F973H         DEFB 00H     ; CQ Putback character
F974H         DEFB 00H     ; RQ Putback character
```

These four variables are used to hold any unwanted character returned to the associated queue. Although the putback facility is implemented in the MSX ROM it is currently unused.

```
F975H VOICAQ: DEFS 128     ; Voice A queue
F9F5H VOICBQ: DEFS 128     ; Voice B queue
FA75H VOICCQ: DEFS 128     ; Voice C queue
FAF5H RS2IQ:  DEFS 64      ; RS232 queue
```

These four buffers contain the three music queues and the RS232 queue, the latter is unused.

```
FB35H PRSCNT: DEFB 00H
```

This variable is used by the "`PLAY`" statement handler to count the number of completed operand strings. Bit 7 is also set after each of the three operands has been parsed to prevent repeated activation of the STRTMS standard routine.

```
FB36H SAVSP: DEFW 0000H
```

This variable is used by the "`PLAY`" statement handler to save the Z80 SP before control transfers to the macro language parser. Its value is compared with the SP on return to determine whether any data has been left on the stack because of a queue-full termination by the parser.

```
FB38H VOICEN: DEFB 00H
```

This variable contains the current voice number being processed by the "`PLAY`" statement handler. The values 0, 1 and 2 correspond to PSG channels A, B and C.

```
FB39H SAVVOL: DEFW 0000H
```

This variable is used by the "`PLAY`" statement "`R`" command handler to save the current volume setting while a zero-amplitude rest is generated.

```
FB3BH MCLLEN: DEFB 00H
```

This variable is used by the macro language parser to hold the length of the string operand being parsed.

```
FB3CH MCLPTR: DEFW 0000H
```

This variable is used by the macro language parser to hold the address of the string operand being parsed.

```
FB3EH QUEUEN: DEFB 00H
```

This variable is used by the interrupt handler to hold the number of the music queue currently being processed. The values 0, 1 and 2 correspond to PSG channels A, B and C.

```
FB3FH MUSICF: DEFB 00H
```

This variable contains three bit flags set by the STRTMS standard routine to initiate processing of a music queue by the interrupt handler. Bits 0, 1 and 2 correspond to VOICAQ, VOICBQ and VOICCQ.

```
FB40H PLYCNT: DEFB 00H
```

This variable is used by the STRTMS standard routine to hold the number of " `PLAY` " statement sequences currently held in the music queues. It is examined when all three end of queue marks have been found for one sequence to determine whether dequeueing should be restarted.

```
FB41H VCBA:   DEFW 0000H   ; Duration counter
FB43H         DEFB 00H     ; String length
FB44H         DEFW 0000H   ; String address
FB46H         DEFW 0000H   ; Stack data address
FB48H         DEFB 00H     ; Music packet length
FB49H         DEFS 7       ; Music packet
FB50H         DEFB 04H     ; Octave
FB51H         DEFB 04H     ; Length
FB52H         DEFB 78H     ; Tempo
FB53H         DEFB 88H     ; Volume
FB54H         DEFW 00FFH   ; Envelope period
FB56H         DEFS 16      ; Space for stack data
```

This thirty-seven byte buffer is used by the " `PLAY` " statement handler to hold the current parameters for voice A.

```
FB66H VCBB:   DEFS 37
```

This buffer is used by the " `PLAY` " statement handler to hold the current parameters for voice B, its structure is the same as VCBA.

```
FB8BH VCBC:   DEFS 37
```

This buffer is used by the " `PLAY` " statement handler to hold the current parameters for voice C, its structure is the same as VCBA.

```
FBB0H ENSTOP: DEFB 00H
```

This variable determines whether the interrupt handler will execute a warm start to the Interpreter upon detecting the keys CODE, GRAPH, CTRL and SHIFT depressed together: 00H=Disable, NZ=Enable.

```
FBB1H BASROM: DEFB 00H
```

This variable determines whether the ISCNTC and INLIN standard routines will respond to the CTRL-STOP key: 00H=Enable, NZ=Disable. It is used to prevent termination of a BASIC ROM located during the power-up ROM search.

```
FBB2H LINTTB: DEFS 24
```

Each of these twenty-four variables is normally non-zero but is zeroed if the contents of the corresponding screen row have overflowed onto the next row. They are maintained by the BIOS but only actually used by the INLIN standard routine (the screen editor) to discriminate between logical and physical lines.

```
FBCAH FSTPOS: DEFW 0000H
```

This variable is used to hold the cursor coordinates upon entry to the INLIN standard routine. Its function is to restrict the extent of backtracking performed when the text is collected from the screen at termination.

```
FBCCH CURSAV: DEFB 00H
```

This variable is used to hold the screen character replaced by the text cursor.

```
FBCDH FNKSWI: DEFB 00H
```

This variable is used by the CHSNS standard routine to determine whether the shifted or unshifted function keys are currently displayed: 00H=Shifted, 01H=Unshifted.

```
FBCEH FNKFLG: DEFS 10
```

Each of these ten variables is normally zero but is set to 01H if the associated function key has been turned on by a "`KEY(n) ON`" statement. They are used by the interrupt handler to determine whether, in program mode only, it should return a character string or update the associated entry in TRPTBL.

```
FBD8H ONGSBF: DEFB 00H
```

This variable is normally zero but is incremented by the interrupt handler whenever a device has achieved the conditions necessary to generate a program interrupt. It is used by the Runloop to determine whether any program interrupts are pending without having to search TRPTBL.

```
FBD9H CLIKFL: DEFB 00H
```

This variable is used internally by the interrupt handler to prevent spurious key clicks when returning multiple characters from a single key depression such as a function key.

```
FBDAH OLDKEY: DEFS 11
```

This buffer is used by the interrupt handler to hold the previous state of the keyboard matrix, each byte contains one row of keys starting with row 0.

```
FBE5H NEWKEY: DEFS 11
```

This buffer is used by the interrupt handler to hold the current state of the keyboard matrix. Key transitions are detected by comparison with the contents of OLDKEY after which OLDKEY is updated with the current state.

```
FBF0H KEYBUF: DEFS 40
```

This buffer contains the decoded keyboard characters produced by the interrupt handler. Note that the buffer is organized as a circular queue driven by GETPNT and PUTPNT and consequently has no fixed starting point.

```
FC18H LINWRK: DEFS 40
```

This buffer is used by the BIOS to hold a complete line of screen characters.

```
FC40H PATWRK: DEFS 8
```

This buffer is used by the BIOS to hold an 8x8 pixel pattern.

```
FC48H BOTTOM: DEFW 8000H
```

This variable contains the address of the lowest RAM location used by the Interpreter. Its value is set at power-up and thereafter unaltered.

```
FC4AH HIMEM:  DEFW F380H
```

This variable contains the address of the byte following the highest RAM location used by the Interpreter. Its value is set at power-up and thereafter only altered by the " `CLEAR` " statement.

```
FC4CH TRPTBL: DEFS 3       ; KEY 1
FC4FH         DEFS 3       ; KEY 2
FC52H         DEFS 3       ; KEY 3
FC55H         DEFS 3       ; KEY 4
FC58H         DEFS 3       ; KEY 5
```

```
FC5BH         DEFS 3       ; KEY 6
FC5EH         DEFS 3       ; KEY 7
FC61H         DEFS 3       ; KEY 8
FC64H         DEFS 3       ; KEY 9
FC67H         DEFS 3       ; KEY 10
FC6AH         DEFS 3       ; STOP
FC6DH         DEFS 3       ; SPRITE
FC70H         DEFS 3       ; STRIG 0
FC73H         DEFS 3       ; STRIG 1
FC76H         DEFS 3       ; STRIG 2
FC79H         DEFS 3       ; STRIG 3
FC7CH         DEFS 3       ; STRIG 4
FC7FH         DEFS 3       ; INTERVAL
FC82H         DEFS 3       ; Unused
FC85H         DEFS 3       ; Unused
FC88H         DEFS 3       ; Unused
FC8BH         DEFS 3       ; Unused
FC8EH         DEFS 3       ; Unused
FC91H         DEFS 3       ; Unused
FC94H         DEFS 3       ; Unused
FC97H         DEFS 3       ; Unused
```

These twenty-six three byte variables hold the current state of the interrupt generating devices. The first byte of each entry contains the device status (bit 0=On, bit 1=Stop, bit 2=Event active) and is updated by the interrupt handler, the Runloop interrupt processor and the "`DEVICE 0=ON/OFF/STOP`" and "`RETURN`" statement handlers. The remaining two bytes of each entry are set by the "`ON DEVICE GOSUB`" statement handler and contain the address of the program line to execute upon a program interrupt.

```
FC9AH RTYCNT: DEFB 00H
```

This variable is unused by the current MSX ROM.

```
FC9BH INTFLG: DEFB 00H
```

This variable is normally zero but is set to 03H or 04H if the CTRL-STOP or STOP keys are detected by the interrupt handler.

```
FC9CH PADY:   DEFB 00H
```

This variable contains the Y coordinate of the last point detected by a touchpad.

```
FC9DH PADX:   DEFB 00H
```

This variable contains the X coordinate of the last point detected by a touchpad.

```
FC9EH JIFFY:  DEFW 0000H
```

This variable is continually incremented by the interrupt handler. Its value may be set or read by the "`TIME`" statement or function.

```
FCA0H INTVAL: DEFW 0000H
```

This variable holds the interval duration set by the "`ON INTERVAL`" statement handler.

```
FCA2H INTCNT: DEFW 0000H
```

This variable is continually decremented by the interrupt handler. When zero is reached its value is reset from INTVAL and, if applicable, a program interrupt generated. Note that this variable always counts irrespective of whether an "`INTERVAL ON`" statement is active.

```
FCA4H LOWLIM: DEFB 31H
```

This variable is used to hold the minimum allowable start bit duration as determined by the TAPION standard routine.

```
FCA5H WINWID: DEFB 22H
```

This variable is used to hold the LO/HI cycle discrimination duration as determined by the TAPION standard routine.

```
FCA6H GRPHED: DEFB 00H
```

This variable is normally zero but is set to 01H by the CNVCHR standard routine upon detection of a graphic header code.

```
FCA7H ESCCNT: DEFB 00H
```

This variable is used by the CHPUT standard routine ESC sequence processor to count escape parameters.

```
FCA8H INSFLG: DEFB 00H
```

This variable is normally zero but is set to FFH by the INLIN standard routine when insert mode is on.

```
FCA9H CSRSW:  DEFB 00H
```

If this variable is zero the cursor is only displayed while the CHGET standard routine is waiting for a keyboard character. If it is non-zero the cursor is permanently displayed via the CHPUT standard routine.

```
FCAAH CSTYLE: DEFB 00H
```

This variable determines the cursor style: 00H=Block, NZ=Underline.

```
FCABH CAPST:  DEFB 00H
```

This variable is used by the interrupt handler to hold the current caps lock status: 00H=Off, NZ=On.

```
FCACH KANAST: DEFB 00H
```

This variable is used to hold the keyboard Kana lock status on Japanese machines and the DEAD key status on European machines.

```
FCADH KANAMD: DEFB 00H
```

This variable holds a keyboard mode on Japanese machines only.

```
FCAEH FLBMEM: DEFB 00H
```

This variable is set by the file I/O error generators but is otherwise unused.

```
FCAFH SCRMOD: DEFB 00H
```

This variable contains the current screen mode: 0=40x24 Text Mode, 1=32x24 Text Mode, 2=Graphics Mode, 3=Multicolour Mode.

```
FCB0H OLDSCR: DEFB 00H
```

This variable holds the screen mode of the last text mode set.

```
FCB1H CASPRV: DEFB 00H
```

This variable is used to hold any character returned to an I/O buffer by the cassette putback function.

```
FCB2H BDRATR: DEFB 00H
```

This variable contains the boundary colour for the "`PAINT`" statement handler. Its value is set by the PNTINI standard routine and used by the SCANR and SCANL standard routines.

```
FCB3H GXPOS:  DEFW 0000H
```

This variable is used for temporary storage of a graphics X coordinate.

```
FCB5H GYPOS:  DEFW 0000H
```

This variable is used for temporary storage of a graphics Y coordinate.

```
FCB7H GRPACX: DEFW 0000H
```

This variable contains the current graphics X coordinate for the GRPPRT standard routine.

```
FCB9H GRPACY: DEFW 0000H
```

This variable contains the current graphics Y coordinate for the GRPPRT standard routine.

```
FCBBH DRWFLG: DEFB 00H
```

Bits 6 and 7 of this variable are set by the "`DRAW`" statement "`N`" and "`B`" command handlers to turn the associated mode on.

```
FCBCH DRWSCL: DEFB 00H
```

This variable is used by the "`DRAW`" statement "`S`" command handler to hold the current scale factor.

```
FCBDH DRWANG: DEFB 00H
```

This variable is used by the "`DRAW`" statement "`A`" command handler to hold the current angle.

```
FCBEH RUNBNF: DEFB 00H
```

This variable is normally zero but is set by the "`BLOAD`" statement handler when an auto-run "`R`" parameter is specified.

```
FCBFH SAVENT: DEFW 0000H
```

This variable contains the "`BSAVE`" and "`BLOAD`" entry address.

```
FCC1H EXPTBL: DEFB 00H     ; Primary Slot 0
FCC2H         DEFB 00H     ; Primary Slot 1
FCC3H         DEFB 00H     ; Primary Slot 2
FCC4H         DEFB 00H     ; Primary Slot 3
```

Each of these four variables is normally zero but is set to 80H during the power-up RAM search if the associated Primary Slot is found to be expanded.

```
FCC5H SLTTBL: DEFB 00H     ; Primary Slot 0
FCC6H         DEFB 00H     ; Primary Slot 1
FCC7H         DEFB 00H     ; Primary Slot 2
FCC8H         DEFB 00H     ; Primary Slot 3
```

These four variables duplicate the contents of the four possible Secondary Slot Registers. The contents of each variable should only be regarded as valid if EXPTBL shows the associated Primary Slot to be expanded.

```
FCC9H SLTATR: DEFS 4       ; PS0, SS0
FCCDH         DEFS 4       ; PS0, SS1
FCD1H         DEFS 4       ; PS0, SS2
FCD5H         DEFS 4       ; PS0, SS3

FCD9H         DEFS 4       ; PS1, SS0
FCDDH         DEFS 4       ; PS1, SS1
FCE1H         DEFS 4       ; PS1, SS2
FCE5H         DEFS 4       ; PS1, SS3

FCE9H         DEFS 4       ; PS2, SS0
FCEDH         DEFS 4       ; PS2, SS1
FCF1H         DEFS 4       ; PS2, SS2
FCF5H         DEFS 4       ; PS2, SS3

FCF9H         DEFS 4       ; PS3, SS0
FCFDH         DEFS 4       ; PS3, SS1
FD01H         DEFS 4       ; PS3, SS2
FD05H         DEFS 4       ; PS3, SS3
```

These sixty-four variables contain the attributes of any extension ROMs found during the power-up ROM search. The characteristics of each 16 KB ROM are encoded into a single byte so four bytes are required for each possible slot. The encoding is:

```
Bit 7 set=BASIC program
Bit 6 set=Device handler
Bit 5 set=Statement handler
```

Note that the entries for page 0 (0000H to 3FFFH) and page 3 (C000H to FFFFH) will always be zero as only page 1 (4000H to 7FFFH) and page 2 (8000H to BFFFH) are actually examined. The MSX convention is that machine code extension ROMs are placed in page 1 and BASIC program ROMs in page 2.

```
FD09H SLTWRK: DEFS 128
```

This buffer provides two bytes of local workspace for each of the sixty-four possible extension ROMs.

```
FD89H PROCNM: DEFS 16
```

This buffer is used to hold a device or statement name for examination by an extension ROM.

```
FD99H DEVICE: DEFB 00H
```

This variable is used to pass a device code, from 0 to 3, to an extension ROM.

The Hooks

The section of the Workspace Area from FD9AH to FFC9H contains one hundred and twelve hooks, each of which is filled with five Z80 RET opcodes at power-up. These are called from strategic locations within the BIOS/Interpreter so that the ROM can be extended, particularly so that it can be upgraded to Disk BASIC. Each hook has sufficient room to hold a far call to any slot:

```
RST 30H
DEFB Slot ID
DEFW Address
RET
```

The hooks are listed on the following pages together with the address they are called from and a brief note as to their function.

ADDRESS	NAME	SIZE	FROM	FUNCTION
FD9AH	HKEYI:	DEFS 5	0C4AH	Interrupt handler
FD9FH	HTIMI:	DEFS 5	0C53H	Interrupt handler
FDA4H	HCHPU:	DEFS 5	08C0H	CHPUT standard routine
FDA9H	HDSPC:	DEFS 5	09E6H	Display cursor
FDAEH	HERAC:	DEFS 5	0A33H	Erase cursor
FDB3H	HDSPF:	DEFS 5	0B2BH	DSPFNK standard routine
FDB8H	HERAF:	DEFS 5	0B15H	ERAFNK standard routine
FDBDH	HTOTE:	DEFS 5	0842H	TOTEXT standard routine
FDC2H	HCHGE:	DEFS 5	10CEH	CHGET standard routine
FDC7H	HINIP:	DEFS 5	071EH	Copy character set to VDP
FDCCH	HKEYC:	DEFS 5	1025H	Keyboard decoder
FDD1H	HKYEA:	DEFS 5	0F10H	Keyboard decoder
FDD6H	HNMI:	DEFS 5	1398H	NMI standard routine
FDDBH	HPINL:	DEFS 5	23BFH	PINLIN standard routine
FDE0H	HQINL:	DEFS 5	23CCH	QINLIN standard routine
FDE5H	HINLI:	DEFS 5	23D5H	INLIN standard routine
FDEAH	HONGO:	DEFS 5	7810H	" `ON DEVICE GOSUB` "
FDEFH	HDSKO:	DEFS 5	7C16H	" `DSKO$` "
FDF4H	HSETS:	DEFS 5	7C1BH	" `SET` "
FDF9H	HNAME:	DEFS 5	7C20H	" `NAME` "
FDFEH	HKILL:	DEFS 5	7C25H	" `KILL` "
FE03H	HIPL:	DEFS 5	7C2AH	" `IPL` "
FE08H	HCOPY:	DEFS 5	7C2FH	" `COPY` "
FE0DH	HCMD:	DEFS 5	7C34H	" `CMD` "
FE12H	HDSKF:	DEFS 5	7C39H	" `DSKF` "
FE17H	HDSKI:	DEFS 5	7C3EH	" `DSKI$` "
FE1CH	HATTR:	DEFS 5	7C43H	" `ATTR$` "

ADDRESS	NAME	SIZE	FROM	FUNCTION
FE21H	HLSET:	DEFS 5	7C48H	" `LSET` "
FE26H	HRSET:	DEFS 5	7C4DH	" `RSET` "
FE2BH	HFIEL:	DEFS 5	7C52H	" `FIELD` "
FE30H	HMKI$:	DEFS 5	7C57H	" `MKI$` "
FE35H	HMKS$:	DEFS 5	7C5CH	" `MKS$` "
FE3AH	HMKD$:	DEFS 5	7C61H	" `MKD$` "
FE3FH	HCVI:	DEFS 5	7C66H	" `CVI` "
FE44H	HCVS:	DEFS 5	7C6BH	" `CVS` "
FE49H	HCVD:	DEFS 5	7C70H	" `CVD` "
FE4EH	HGETP:	DEFS 5	6A93H	Locate FCB
FE53H	HSETF:	DEFS 5	6AB3H	Locate FCB
FE58H	HNOFO:	DEFS 5	6AF6H	" `OPEN` "
FE5DH	HNULO:	DEFS 5	6B0FH	" `OPEN` "
FE62H	HNTFL:	DEFS 5	6B3BH	Close I/O buffer 0
FE67H	HMERG:	DEFS 5	6B63H	" `MERGE/LOAD` "
FE6CH	HSAVE:	DEFS 5	6BA6H	" `SAVE` "
FE71H	HBINS:	DEFS 5	6BCEH	" `SAVE` "
FE76H	HBINL:	DEFS 5	6BD4H	" `MERGE/LOAD` "
FE7BH	HFILE:	DEFS 5	6C2FH	" `FILES` "
FE80H	HDGET:	DEFS 5	6C3BH	" `GET/PUT` "
FE85H	HFILO:	DEFS 5	6C51H	Sequential output
FE8AH	HINDS:	DEFS 5	6C79H	Sequential input
FE8FH	HRSLF:	DEFS 5	6CD8H	" `INPUT$` "
FE94H	HSAVD:	DEFS 5	6D03H, 6D14H	" `LOC` ", " `LOF` ",
			6D25H, 6D39H	" `EOF` ", " `FPOS` "
FE99H	HLOC:	DEFS 5	6D0FH	" `LOC` "
FE9EH	HLOF:	DEFS 5	6D20H	" `LOF` "

ADDRESS	NAME	SIZE	FROM	FUNCTION
FEA3H	HEOF:	DEFS 5	6D33H	" `EOF` "
FEA8H	HFPOS:	DEFS 5	6D43H	" `FPOS` "
FEADH	HBAKU:	DEFS 5	6E36H	" `LINE INPUT#` "
FEB2H	HPARD:	DEFS 5	6F15H	Parse device name
FEB7H	HNODE:	DEFS 5	6F33H	Parse device name
FEBCH	HPOSD:	DEFS 5	6F37H	Parse device name
FEC1H	HDEVN:	DEFS 5		This hook is not used.
FEC6H	HGEND:	DEFS 5	6F8FH	I/O function dispatcher
FECBH	HRUNC:	DEFS 5	629AH	Run-clear
FED0H	HCLEA:	DEFS 5	62A1H	Run-clear
FED5H	HLOPD:	DEFS 5	62AFH	Run-clear
FEDAH	HSTKE:	DEFS 5	62F0H	Reset stack
FEDFH	HISFL:	DEFS 5	145FH	ISFLIO standard routine
FEE4H	HOUTD:	DEFS 5	1B46H	OUTDO standard routine
FEE9H	HCRDO:	DEFS 5	7328H	CR,LF to OUTDO
FEEEH	HDSKC:	DEFS 5	7374H	Mainloop line input
FEF3H	HDOGR:	DEFS 5	593CH	Line draw
FEF8H	HPRGE:	DEFS 5	4039H	Program end
FEFDH	HERRP:	DEFS 5	40DCH	Error handler
FF02H	HERRF:	DEFS 5	40FDH	Error handler
FF07H	HREAD:	DEFS 5	4128H	Mainloop " `OK` "
FF0CH	HMAIN:	DEFS 5	4134H	Mainloop
FF11H	HDIRD:	DEFS 5	41A8H	Mainloop direct statement
FF16H	HFINI:	DEFS 5	4237H	Mainloop finished
FF1BH	HFINE:	DEFS 5	4247H	Mainloop finished
FF20H	HCRUN:	DEFS 5	42B9H	Tokenize
FF25H	HCRUS:	DEFS 5	4353H	Tokenize

ADDRESS	NAME	SIZE	FROM	FUNCTION
FF2AH	HISRE:	DEFS 5	437CH	Tokenize
FF2FH	HNTFN:	DEFS 5	43A4H	Tokenize
FF34H	HNOTR:	DEFS 5	44EBH	Tokenize
FF39H	HSNGF:	DEFS 5	45D1H	" `FOR` "
FF3EH	HNEWS:	DEFS 5	4601H	Runloop new statement
FF43H	HGONE:	DEFS 5	4646H	Runloop execute
FF48H	HCHRG:	DEFS 5	4666H	CHRGTR standard routine
FF4DH	HRETU:	DEFS 5	4821H	" `RETURN` "
FF52H	HPRTF:	DEFS 5	4A5EH	" `PRINT` "
FF57H	HCOMP:	DEFS 5	4A54H	" `PRINT` "
FF5CH	HFINP:	DEFS 5	4AFFH	" `PRINT` "
FF61H	HTRMN:	DEFS 5	4B4DH	" `READ/INPUT` " error
FF66H	HFRME:	DEFS 5	4C6DH	Expression Evaluator
FF6BH	HNTPL:	DEFS 5	4CA6H	Expression Evaluator
FF70H	HEVAL:	DEFS 5	4DD9H	Factor Evaluator
FF75H	HOKNO:	DEFS 5	4F2CH	Factor Evaluator
FF7AH	HFING:	DEFS 5	4F3EH	Factor Evaluator
FF7FH	HISMI:	DEFS 5	51C3H	Runloop execute
FF84H	HWIDT:	DEFS 5	51CCH	" `WIDTH` "
FF89H	HLIST:	DEFS 5	522EH	" `LIST` "
FF8EH	HBUFL:	DEFS 5	532DH	Detokenize
FF93H	HFRQI:	DEFS 5	543FH	Convert to integer
FF98H	HSCNE:	DEFS 5	5514H	Line number to pointer
FF9DH	HFRET:	DEFS 5	67EEH	Free descriptor
FFA2H	HPTRG:	DEFS 5	5EA9H	Variable search
FFA7H	HPHYD:	DEFS 5	148AH	PHYDIO standard routine
FFACH	HFORM:	DEFS 5	148EH	FORMAT standard routine

ADDRESS	NAME	SIZE	FROM	FUNCTION
FFB1H	HERRO:	DEFS 5	406FH	Error handler
FFB6H	HLPTO:	DEFS 5	085DH	LPTOUT standard routine
FFBBH	HLPTS:	DEFS 5	0884H	LPTSTT standard routine
FFC0H	HSCRE:	DEFS 5	79CCH	" `SCREEN` "
FFC5H	HPLAY:	DEFS 5	73E5H	" `PLAY` " statement

The Workspace Area from FFCAH to FFFFH is unused. (on MSX 1)

7. Machine Code Programs

This chapter contains a number of machine code programs to illustrate the use of MSX system resources. Although prepared with the ZEN Assembler they are designed t o run from BASIC and if necessary, may be entered in hex form using the loader shown below. The code should then be saved on cassette before any attempt is made to run it.

```
10 CLEAR 200,&HE000
20 ADDR=&HE000
30 PRINT RIGHT$ ("000"+HEX$(ADDR),4);
40 INPUT D$
50 POKE ADDR,VAL("&H"+D$)
60 ADDR=ADDR+1
70 GOTO 30
```

All the programs start at address E000H and are entered at the same point. Unless stated otherwise no parameter need be passed to a program, execution may therefore be initiated with a simple `DEFUSR=&HE000:?USR(0)` statement.

Keyboard Matrix

This program displays the keyboard matrix on the screen so that key depressions may be directly observed. The program may be terminated by pressing the CTRL and STOP keys. Note that spurious key depressions can be produced under certain circumstances if more than three or four keys are pressed at one time. This is a characteristic of all matrix type keyboards.

```
                        ORG     0E000H
                        LOAD    0E000H

                    ; ****************************
                    ; *   BIOS STANDARD ROUTINES   *
                    ; ****************************

                    INITXT: EQU     006CH
                    CHPUT:  EQU     00A2H
                    SNSMAT: EQU     0141H
                    BREAKX: EQU     00B7H

                    ; ****************************
                    ; *     WORKSPACE VARIABLES    *
                    ; ****************************

                    INTFLG: EQU     0FC9BH

                    ; ****************************
                    ; *      CONTROL CHARACTERS    *
                    ; ****************************

                    LF:     EQU     10
                    HOME:   EQU     11
                    CR:     EQU     13

E000    CD6C00      MATRIX: CALL    INITXT          ; SCREEN 0
E003    3E0B        MX1:    LD      A,HOME          ;
E005    CDA200              CALL    CHPUT           ; Home Cursor
E008    AF                  XOR     A               ; A=KBD row
E009    F5          MX2:    PUSH    AF              ;
E00A    CD4101              CALL    SNSMAT          ; Read a row
E00D    0608                LD      B,6             ; Eight cols
E00F    07          MX3:    RLCA                    ; Select col
E010    F5                  PUSH    AF              ;
E011    E601                AND     1               ;
E013    C630                ADD     A,"0"           ; Result
E015    CDA200              CALL    CHPUT           ; Display col
E018    F1                  POP     AF              ;
E019    10F4                DJNZ    MX3             ;
E01B    3E0D                LD      A,CR            ; Newline
E01D    CDA200              CALL    CHPUT           ;
E020    3E0A                LD      A,LF            ;
E022    CDA200              CALL    CHPUT           ;
E025    F1                  POP     AF              ; A=KBD row
E026    3C                  INC     A               ; Next row
E027    FE0B                CP      11              ; Finished?
E029    20DE                JR      NZ,MX2          ;
E02B    CDB700              CALL    BREAKX          ; CTRL-STOP
E02E    30D3                JR      NC,MX1          ; Continue
E030    AF                  XOR     A               ;
E031    329BFC              LD      (INTFLG),A      ; Clear possible STOP
```

```
E034    C9                  RET                         ; Back to BASIC

                            END
```

40 Column Graphics Text

This program prints text on the Graphics Mode screen at forty characters per line. The string to be displayed is passed as the `USR` call parameter, for example `A$=USR("something")`. There us no need to open a GRP file beforehand, the only requirement of the program is that the screen be in the correct mode. The heart of the program is functionally equivalent to the GRPPRT standard routine but only the first six dot columns of a given character pattern are placed on the screen instead of eight. As the GRPPRT the pattern is placed at the current graphics position and the only control character recognised is ASCII CR (13) which functions as a combined CR, LF. Unlike the GRPPRT standard routine characters printed at negative coordinates, but which overlap the screen, will be correctly displayed. The program is currently set up to perform an auto linefeed after dot column 239, thus giving exactly forty characters per line. If required this may be changed, via the constant in the RMDCOL subroutine, so that the full width of the screen is usable.

```
                ORG     0E000H
                LOAD    0E000H

; ******************************
; *   BIOS STANDARD ROUTINES   *
; ******************************

RDSLT:  EQU     000CH
CNVCHR: EQU     00ABH
MAPXYC: EQU     0111H
SETC:   EQU     0120H

; ******************************
; *     WORKSPACE VARIABLES    *
; ******************************

FORCLR: EQU     0F3E9H
ATRBYT: EQU     0F3F2H
CGPNT:  EQU     0F91FH
PATWRK: EQU     0FC40H
SCRMOD: EQU     0FCAFH
GRPACX: EQU     0FCB7H
GRPACY: EQU     0FCB9H

; ******************************
; *      CONTROL CHARACTERS    *
; ******************************

CR:     EQU     13
```

```
E000    FE03        GFORTY: CP      3               ; String type?
E002    C0                  RET     NZ              ;
E003    3AAFFC              LD      A,(SCRMOD)      ; Mode
E006    FE02                CP      2               ; Graphics?
E008    C0                  RET     NZ              ;
E009    EB                  EX      DE,HL           ; HL->Descriptor
E00A    46                  LD      B,(HL)          ; B=String len
E00B    23                  INC     HL              ;
E00C    5E                  LD      E,(HL)          ; Address LSB
E00D    23                  INC     HL              ;
E00E    56                  LD      D,(HL)          ; DE->String
E00F    04                  INC     B               ;
E010    05          GF2:    DEC     B               ; Finished?
E011    C8                  RET     Z               ;
E012    1A                  LD      A,(DE)          ; A=Chr from string
E013    CD19E0              CALL    GPRINT          ; Print it
E016    13                  INC     DE              ;
E017    18F7                JR      GF2             ; Next chr
E019    F5          GPRINT: PUSH    AF              ;
E01A    C5                  PUSH    BC              ;
E01B    D5                  PUSH    DE              ;
E01C    E5                  PUSH    HL              ;
E01D    FDE5                PUSH    IY              ;
E01F    ED4BB7FC            LD      BC,(GRPACX)     ; BC=X coord
E023    ED5BB9FC            LD      DE,(GRPACY)     ; DE=Y coord
E027    CD39E0              CALL    GDC             ; Decode chr
E02A    ED43B7FC            LD      (GRPACX),BC     ; New X coord
E02E    ED53B9FC            LD      (GRPACY),DE     ; New Y coord
E032    FDE1                POP     IY              ;
E034    E1                  POP     HL              ;
E035    D1                  POP     DE              ;
E036    C1                  POP     BC              ;
E037    F1                  POP     AF              ;
E038    C9                  RET                     ;

E039    CDAB00      GDC:    CALL    CNVCHR          ; Check graphic
E03C    D0                  RET     NC              ; NC=Header
E03D    2007                JR      NZ,GD2          ; NZ=Converted
E03F    FE0D                CP      CR              ; Carriage Return?
E041    2873                JR      Z,GCRLF         ;
E043    FE20                CP      20H             ; Other control?
E045    D8                  RET     C               ; Ignore
E046    6F          GD2:    LD      L,A             ;
E047    2600                LD      H,0             ; HL=Chr code
E049    29                  ADD     HL,HL           ;
E04A    29                  ADD     HL,HL           ;
E04B    29                  ADD     HL,HL           ; HL=Chr*8
E04C    C5                  PUSH    BC              ; X coord
E04D    D5                  PUSH    DE              ; Y coord
E04E    ED5B20F9            LD      DE,(CGPNT+1)    ; Character set
E052    19                  ADD     HL,DE           ; HL->Pattern
E053    1140FC              LD      DE,PATWRK       ; DE->Buffer
E056    0608                LD      B,8             ; Eight byte pattern
```

```
E058    C5          GD3:    PUSH    BC              ;
E059    D5                  PUSH    DE              ;
E05A    3A1FF9              LD      A,(CGPNT)       ; Slot ID
E05D    CD0C00              CALL    RDSLT           ; Get pattern
E060    FB                  EI                      ;
E061    D1                  POP     DE              ;
E062    C1                  POP     BC              ;
E063    12                  LD      (DE),A          ; Put in buffer
E064    13                  INC     DE              ;
E065    23                  INC     HL              ;
E066    10F0                DJNZ    GD3             ; Next
E068    D1                  POP     DE              ;
E069    C1                  POP     BC              ;
E06A    3AE9F3              LD      A,(FORCLR)      ; Current colour
E06D    32F2F3              LS      (ATRBYT),A      ; Set ink
E070    FD2140FC            LD      IY,PATWRK       ; IY->Patterns
E074    D5                  PUSH    DE              ;
E075    2608                LD      H,8             ; Max dot rows
E077    CB7A        GD4:    BIT     7,D             ; Pos Y coord?
E079    202A                JR      NZ,GD8          ;
E07B    CDBFE0              CALL    BMDROW          ; Bottom most row?
E07E    382B                JR      C,GD9           ; C=Y too large
E080    C5                  PUSH    BC              ;
E081    2E06                LD      L,6             ; Max dot cols
E083    FD7E00              LD      A,(IY+0)        ; A=Pattern row
E086    CB78        GD5:    BIT     7,B             ; Pos X coord
E088    2015                JR      NZ,GD6          ;
E08A    CDC8E0              CALL    RMDCOL          ; Rightmost col?
E08D    3815                JR      C,GD7           ; C=X too large
E08F    CB7F                BIT     7,A             ; Pattern bit
E091    280C                JR      Z,GD6           ; Z=0 Pixel
E093    F5                  PUSH    AF              ;
E094    D5                  PUSH    DE              ;
E095    E5                  PUSH    HL              ;
E096    CD1101              CALL    MAPXYC          ; Map coords
E099    CD2001              CALL    SETC            ; Set pixel
E09C    E1                  POP     HL              ;
E09D    D1                  POP     DE              ;
E09E    F1                  POP     AF              ;
E09F    07          GD6:    RLCA                    ; Shift pattern
E0A0    03                  INC     BC              ; X=X+1
E0A1    2D                  DEC     L               ; Finished dot cols?
E0A2    20E2                JR      NZ,GD5          ;
E0A4    C1          GD7:    POP     BC              ; Initial X coord
E0A5    FD23        GD8:    INC     IY              ; Next pattern byte
E0A7    13                  INC     DE              ; Y=Y+1
E0A8    25                  DEC     H               ; Finished dot rows?
E0A9    20CC                JR      NZ,GD4          ;
E0AB    D1          GD9:    POP     DE              ; Initial Y coord
E0AC    210600              LD      HL,6            ; Step
E0AF    09                  ADD     HL,BC           ; X=X+6
E0B0    44                  LD      B,H             ;
E0B1    4D                  LD      C,L             ; BC=New X coord
```

```
E0B2    CDC8E0              CALL    RMDCOL          ; Rightmost col?
E0B5    D0                  RET     NC              ;

E0B6    010000      GCRLF:  LD      BC,0            ; X=0
E0B9    210800              LD      HL,8            ;
E0BC    19                  ADD     HL,DE           ;
E0BD    EB                  EX      DE,HL           ; Y=Y+8
E0BE    C9                  RET                     ;

E0BF    E5          BMDROW: PUSH    HL              ;
E0C0    21BF00              LD      HL,191          ; Bottom dot row
E0C3    B7                  OR      A               ;
E0C4    ED52                SBC     HL,DE           ; Check Y coord
E0C6    E1                  POP     HL              ;
E0C7    C9                  RET                     ; C=Below screen

E0C8    E5          RMDCOL: PUSH    HL              ;
E0C9    21EF00              LD      HL,239          ; Rightmost dot col
E0CC    B7                  OR      A               ;
E0CD    ED42                SBC     HL,BC           ; Check X coord
E0CF    E1                  POP     HL              ;
E0D0    C9                  RET                     ; C=Beyond right

                            END
```

String Bubble Sort

This program will sort the contents os a string Array into ascending alphabetic order. The location of the Array is passed as the `USR` call parameter, for example `V=USR(VARPRT(A$(0)))`. There are no restrictions on the size of the Array or on its contents but it must only have one dimension. The program is based on the classic bubble sort algorithm where string pairs are compared and their positions swapped if the second is smaller than the first. a 250 element Array of randomly generated stringswill be sorted in approximately 2.5 seconds. The equivalent BASIC program takes over six minutes.

```
                            ORG     0E000H
                            LOAD    0E000H

E000    FE02        SORT:   CP      2               ; Integer type?
E002    C0                  RET     NZ              ;
E003    23                  INC     HL              ; HL->DAC+1
E004    23                  INC     HL              ; HL->DAC+2
E005    5E                  LD      E,(HL)          ; Address LSB
E006    23                  INC     HL              ; HL->DAC+3
E007    56                  LD      D,(HL)          ; Address MSB
E008    EB                  EX      DE,HL           ; HL->A$(0)
E009    E5                  PUSH    HL              ;
E00A    DDE1                POP     IX              ; IX->A$(0)
E00C    DD7EF8              LD      A,(IX-8)        ; Array type
```

```
E00F    FE03            CP      3               ; String Array?
E011    C0              RET     NZ              ;
E012    DD7EFD          LD      A,(IX-3)        ; Dimension
E015    3D              DEC     A               ; Single dimension?
E016    C0              RET     NZ              ;
E017    DD4EFE          LD      C,(IX-2)        ;
E01A    DD46FF          LD      B,(IX-1)        ; BC=Element count
E01D    C5      SR2:    PUSH    BC              ;
E01E    E5              PUSH    HL              ; HL->Dsc(N)
E01F    46      SR3:    LD      B,(HL)          ; B=Len(N)
E020    23              INC     HL              ;
E021    5E              LD      E,(HL)          ;
E022    23              INC     HL              ;
E023    E5              PUSH    HL              ;
E024    56              LD      D,(HL)          ; DE->String(N)
E025    23              INC     HL              ; HL->Dsc(N+1)
E026    4E              LD      C,(HL)          ; C=Len(N+1)
E027    23              INC     HL              ;
E028    7E              LD      A,(HL)          ;
E029    23              INC     HL              ;
E02A    E5              PUSH    HL              ;
E02B    66              LD      H,(HL)          ;
E02C    6F              LD      L,A             ; HL->String(N+1)
E02D    EB              EX      DE,HL           ; HL->(N),DE->(N+1)
E02E    04              INC     B               ;
E02F    0C              INC     C               ;
E030    05      SR4:    DEC     B               ; Remaining len(N)
E031    2B25            JR      Z,NEXT          ; Z=(N)<=(N+1)
E033    0D              DEC     C               ; Remaining len(N+1)
E034    2808            JR      Z,SWAP          ; Z=(N+1)<(N)
E036    1A              LD      A,(DE)          ; Chr from (N+1)
E037    BE              CP      (HL)            ; Chr from (N)
E038    13              INC     DE              ;
E039    23              INC     HL              ;
E03A    28F4            JR      Z,SR4           ; Same, continue
E03C    301A            JR      NC,NEXT         ; NC=(N)<(N+1)
E03E    E1      SWAP:   POP     HL              ; HL->Dsc(N+1)
E03F    D1              POP     DE              ; DE->Dsc(N)
E040    0603            LD      B,3             ; Descriptor size
E042    1A      SW2:    LD      A,(DE)          ; Swap descriptors
E043    4E              LD      C,(HL)          ;
E044    77              LD      (HL),A          ;
E045    79              LD      A,C             ;
E046    12              LD      (DE),A          ;
E047    1B              DEC     DE              ;
E048    2B              DEC     HL              ;
E049    10F7            DJNZ    SW2             ;
E04B    DDE5            PUSH    IX              ;
E04D    E1              POP     HL              ; HL->A$(0)
E04E    B7              OR      A               ;
E04F    ED52            SBC     HL,DE           ; At Array start?
E051    3007            JR      NC,NX2          ; NC=At start
E053    1B              DEC     DE              ; Back up
```

```
E054    1B              DEC     DE              ;
E055    EB              EX      DE,HL           ; HL->Dsc(N-1_
E056    18C7            JR      SR3             ; Go check again
E058    E1      NEXT:   POP     HL              ; Lose junk
E059    E1              POP     HL              ;
E05A    E1      NX2:    POP     HL              ; HL->Dsc(N)
E05B    C1              POP     BC              ; BC=Element count
E05C    23              INC     HL              ; Next descriptor
E05D    23              INC     HL              ;
E05E    23              INC     HL              ;
E05F    0B              DEC     BC              ;
E060    78              LD      A,B             ;
E061    B1              OR      C               ; Finished?
E062    20B9            JR      NZ,SR2          ;
E064    C9              RET                     ;

                        END
```

Graphics Screen Dump

This program will dump the screen contents, in any mode, to the printer. When first activated via a `USR` call the program merely patches itself into the interrupt handler keyscan hook.

Once the program has installed itself it effectively becomes an extension of the interrupt handler and a screen dump may then be initiated from any part of the system simply by pressing the ESC key. If necessary the dump can be terminated by pressing the CTRL and STOP keys. An example of a Graphics Mode screen, in which all thirty-two sprites are active, is shown below:

The simplest method of generating a screen dump is to copy all the character codes from the Name Table to the printer. However this would only work in the two text modes, the sprites could not be displayed and the result would reflect the printer's internal character set rather than the VDP character set. The program therefore reproduces the screen as a 240/256x192 bit image on the printer in all modes, each point in the image being derived from the colour code of the corresponding point on the screen. No dot for colours 0 to 7 and a dot for colours 8 to 15.

The colour code for a given point is obtained by first examining the thirty-two sprites in sequence to determine whether any one overlaps it. If every sprite is transparent at the point then the character plane is examined. This is done by using the point coordinates to locate the corresponding entry in the Name Table then, via the character code, to isolate the relevant bit in the associated pattern. If the bit's colour code is found to be transparent the background plane colour is returned.

Note that the control code sequences used in the program are the Epson FX80 printer. These are marked in the listings in case another printer is to be used. One sequence is used to enter bit image mode at the start of a 240/256 byte line (each byte defines eight vertical dots) and one sequence is used to initiate a paper feed at the end of the line. The program is generally optimised for speed, rather than for minimal code, and takes about five seconds plus printer time to produce the 46,080/49,152 dots in the image.

```
                        ORG     0E000H
                        LOAD    0E000H

                    ; ****************************
                    ; *   BIOS STANDARD ROUTINES   *
                    ; ****************************

                    RDVRM:  EQU     004AH
                    CALATR: EQU     0087H
                    LPTOUT: EQU     00A5H

                    ; ****************************
                    ; *     WORKSPACE VARIABLES    *
                    ; ****************************

                    T32COL: EQU     0F3BFH
                    GRPNAM: EQU     0F3C7H
                    GRPCOL: EQU     0F3C9H
                    GRPCGP: EQU     0F3CBH
                    MLTNAM: EQU     0F3D1H
                    MLTCGP: EQU     0F3D5H
                    RG1SAV: EQU     0F3E0H
                    RG7SAV: EQU     0F3E6H
                    NAMBAS: EQU     0F922H
                    CGPBAS: EQU     0F924H
                    PATBAS: EQU     0F926H
                    ATRBAS: EQU     0F928H
                    SCRMOD: EQU     0FCAFH
                    HKEYC:  EQU     0FDCCH

                    ; ****************************
                    ; *      CONTROL CHARACTERS    *
                    ; ****************************

                    CR:     EQU     13
                    ESC:    EQU     27

E000    3ACCFD      ENTRY:  LD      A,(HKEYC)           ; Hook
E003    FEC9                CP      0C9H                ; Free to use?
E005    C0                  RET     NZ                  ;
E006    2112E0              LD      HL,DUMP             ; Where to go
E009    22CDFD              LD      (HKEYC+1),HL        ; Redirect hook
E00C    3ECD                LD      A,0CDH              ; CALL
E00E    32CCFD              LD      (HKEYC),A           ;
E011    C9                  RET                         ;

E012    FE3A        DUMP:   CP      3AH                 ; ESC key number?
E014    C0                  RET     NZ                  ;
E015    F5                  PUSH    AF                  ;
E016    C5                  PUSH    BC                  ;
E017    D5                  PUSH    DE                  ;
E018    E5                  PUSH    HL                  ;
```

```
E019    ED734FE2            LD      (BRKSTK),SP         ; For CTRL-STOP
E01D    0E00                LD      C,0                 ; C=Row
E01F    3AAFFC      DU1:    LD      A,(SCRMOD)          ; Mode
E022    B7                  OR      A                   ;
E023    21F000              LD      HL,240              ; T40 Dots per row
E026    112B06              LD      DE,6*256+40         ;
E029    2806                JR      Z,DU2               ;
E02B    210001              LD      HL,256              ; T32,GRP,MLT Dots
E02E    112008              LD      DE,8*256+32         ;
E031    3E1B        DU2:    LD      A,ESC               ; ***** FX80 *****
E033    CD8DE0              CALL    PRINT               ; *              *
E036    3E4B                LD      A,"K"               ; *   Bit mode   *
E038    CD8DE0              CALL    PRINT               ; *              *
E03B    7D                  LD      A,L                 ; *  Bytes  LSB  *
E03C    CD8DE0              CALL    PRINT               ; *              *
E03F    7C                  LD      A,H                 ; *  Bytes  MSB  *
E040    CD8DE0              CALL    PRINT               ; ****************
E043    0600                LD      B,0                 ; B=Column
E045    CD97E0      DU3:    CALL    CELL                ; Do an 8x8 cell
E048    D5                  PUSH    DE                  ;
E049    C5                  PUSH    BC                  ;
E04A    2151E2              LD      HL,CBUFF            ; HL->Colours
E04D    42                  LD      B,D                 ; B=Dot cols (6 or 8)
E04E    110800              LD      DE,8                ; CBUFF offset
E051    C5          DU4:    PUSH    BC                  ;
E052    E5                  PUSH    HL                  ;
E053    0608                LD      B,8                 ; B=Dot rows
E055    7E          DU5:    LD      A,(HL)              ; A=Colour code
E056    FE08                CP      8                   ; Dark or light?
E058    3F                  CCF                         ; Light=Print dot
E059    CB11                RL      C                   ; Build result
E05B    19                  ADD     HL,DE               ; Next dot row
E05C    10F7                DJNZ    DU5                 ;
E05E    79                  LD      A,C                 ; 8 Vertical dots
E05F    CD8DE0              CALL    PRINT               ;
E062    E1                  POP     HL                  ;
E063    C1                  POP     BC                  ;
E064    23                  INC     HL                  ; Next dot col
E065    10EA                DJNZ    DU4                 ;
E067    C1                  POP     BC                  ;
E068    D1                  POP     DE                  ;
E069    04                  INC     B                   ; Next column
E06A    78                  LD      A,B                 ;
E06B    BB                  CP      E                   ; End of row?
E06C    20D7                JR      NZ,DU3              ;
E06E    3E0D                LD      A,CR                ; Head left
E070    CD8DE0              CALL    PRINT               ;
E073    3E1B                LD      A,ESC               ; ***** FX80 *****
E075    CD8DE0              CALL    PRINT               ; *              *
E078    3E4A                LD      A,"J"               ; *  Paper feed  *
E07A    CD8DE0              CALL    PRINT               ; *              *
E07D    3E18                LD      A,24                ; * 24/216= 1/9" *
E07F    CD8DE0              CALL    PRINT               ; ****************
```

```
E082    0C                  INC     C               ; Next row
E083    79                  LD      A,C             ;
E084    FE18                CP      24              ; Finished screen?
E086    2097                JR      NZ,DU1          ;
E088    E1          DU6:    POP     HL              ;
E089    D1                  POP     DE              ;
E08A    C1                  POP     BC              ;
E08B    F1                  POP     AF              ;
E08C    C9                  RET                     ;

E08D    CDA500      PRINT:  CALL    LPTOUT          ; To printer
E090    D0                  RET     NC              ; CTRL-STOP?
E091    ED7B4FE2            LD      SP,(BRKSTK)     ; Restore stack
E095    18F1                JR      DU6             ; Terminate program

E097    C5          CELL:   PUSH    BC              ;
E098    D5                  PUSH    DE              ;
E099    E5                  PUSH    HL              ;
E09A    FDE5                PUSH    IY              ;
E09C    2151E2              LD      HL,CBUFF        ; For results
E09F    3E40                LD      A,64            ;
E0A1    3600        CL1:    LD      (HL),0          ; Transparent
E0A3    23                  INC     HL              ;
E0A4    3D                  DEC     A               ; Fill
E0A5    20FA                JR      NZ,CL1          ;
E0A7    3AAFFC              LD      A,(SCRMOD)      ; Mode
E0AA    B7                  OR      A               ; T40?
E0AB    F5                  PUSH    AF              ;
E0AC    C5                  PUSH    BC              ;
E0AD    C469E1              CALL    NZ,SPRTES       ; Sprites first
E0B0    C1                  POP     BC              ;
E0B1    69                  LD      L,C             ;
E0B2    2600                LD      H,0             ; HL=Row
E0B4    29                  ADD     HL,HL           ;
E0B5    29                  ADD     HL,HL           ;
E0B6    29                  ADD     HL,HL           ; HL=Row*8
E0B7    5D                  LD      E,L             ;
E0B8    54                  LD      D,H             ; DE=Row*8
E0B9    29                  ADD     HL,HL           ;
E0BA    29                  ADD     HL,HL           ; HL=Row*32
E0BB    F1                  POP     AF              ; Mode
E0BC    F5                  PUSH    AF              ;
E0BD    2001                JR      NZ,CL2          ; T40?
E0BF    19                  ADD     HL,DE           ; HL=Row*40
E0C0    58          CL2:    LD      E,B             ; DE=Column
E0C1    19                  ADD     HL,DE           ;
E0C2    EB                  EX      DE,HL           ; DE=NAMTAB offset
E0C3    D602                SUB     2               ; Mode
E0C5    79                  LD      A,C             ; A=Row
E0C6    010000              LD      BC,0            ; BC=CGPTAB offset
E0C9    2A24F9              LD      HL,(CGPBAS)     ;
E0CC    E5                  PUSH    HL              ;
E0CD    2A22F9              LD      HL,(NAMBAS)     ;
```

```
E0D0    3819            JR      C,CL4           ; C=T40 or T32
E0D2    200C            JR      NZ,CL3          ; NZ=MLT
E0D4    2ACBF3          LD      HL,(GRPCGP)     ; Else GRP
E0D7    E3              EX      (SP),HL         ;
E0D8    2AC7F3          LD      HL,(GRPNAM)     ;
E0DB    E618            AND     18H             ; Row MSBs
E0DD    47              LD      B,A             ; 1/3=2kB CGP offset
E0DE    180B            JR      CL4             ;
E0E0    2AD5F3  CL3:    LD      HL,(MLTCGP)     ;
E0E3    E3              EX      (SP),HL         ;
E0E4    2AD1F3          LD      HL,(MLTNAM)     ;
E0E7    07              RLCA                    ; Row*2
E0E8    E606            AND     6               ;
E0EA    4F              LD      C,A             ; 1/6=2B CGP offset
E0EB    19      CL4:    ADD     HL,DE           ; HL->NAMTAB
E0EC    CD4A00          CALL    RDVRM           ; Get chr code
E0EF    6F              LD      L,A             ;
E0F0    2600            LD      H,0             ; HL=Chr code
E0F2    29              ADD     HL,HL           ;
E0F3    29              ADD     HL,HL           ;
E0F4    29              ADD     HL,HL           ; HL=Chr*8
E0F5    09              ADD     HL,BC           ; GRP,MLT offsets
E0F6    EB              EX      DE,HL           ; DE=CGPTAB offset
E0F7    FDE1            POP     IY              ; IY=CGPTAB base
E0F9    FD19            ADD     IY,DE           ; IY->Pattern
E0FB    2AC9F3          LD      HL,(GRPCOL)     ;
E0FE    19              ADD     HL,DE           ; HL->GRP colours
E0FF    0F              RRCA                    ;
E100    0F              RRCA                    ;
E101    0F              RRCA                    ; Chr code/8
E102    E61F            AND     1FH             ;
E104    4F              LD      C,A             ;
E105    0600            LD      B,0             ;
E107    3AE6F3          LD      A,(RG7SAV)      ; T40 Colours
E10A    57              LD      D,A             ; D=T40 Colours
E10B    E60F            AND     0FH             ;
E10D    5F              LD      E,A             ; E=Background colour
E10E    F1              POP     AF              ; Mode
E10F    E5              PUSH    HL              ; STK->GRP Colours
E110    3D              DEC     A               ;
E111    2008            JR      NZ,CL5          ; Z=T32
E113    2ABFF3          LD      HL,(T32COL)     ;
E116    09              ADD     HL,BC           ; HL->T32 Colours
E117    CD4A00          CALL    RDVRM           ; Get T32 Colours
E11A    57              LD      D,A             ; D=T32 Colours
E11B    2151E2  CL5:    LD      HL,CBUFF        ; Results
E11E    0608            LD      B,8             ; Dot rows
E120    FDE5    CL6:    PUSH    IY              ;
E122    E3              EX      (SP),HL         ; HL->Pattern
E123    CD4A00          CALL    RDVRM           ; Get pattern
E126    4F              LD      C,A             ; C=Pattern
E127    E1              POP     HL              ;
E128    FD23            INC     IY              ; Next dot row
```

```
E12A    3AAFFC              LD      A,(SCRMOD)          ; Mode
E12D    D602                SUB     2                   ;
E12F    3815                JR      C,CL8               ; C=T40 or T32
E131    280C                JR      Z,CL7               ; Z=GRP
E133    51                  LD      D,C                 ; MLT Colours=Pattern
E134    0EF0                LD      C,0F0H              ; Dummy MLT pattern
E136    78                  LD      A,B                 ; Dot row
E137    FE05                CP      5                   ; Cell halfway mark?
E139    280B                JR      Z,CL8               ;
E13B    FD2B                DEC     IY                  ; Back up pattern
E13D    1807                JR      CL8                 ;
E13F    E3          CL7:    EX      (SP),HL             ; HL->GRP Colours
E140    CD4A00              CALL    RDVRM               ; Get colours
E143    57                  LD      D,A                 ; D=GRP Colours
E144    23                  INC     HL                  ; Next dot row
E145    E3                  EX      (SP),HL             ; STK->GRP Colours
E146    C5          CL8:    PUSH    BC                  ;
E147    0608                LD      B,8                 ; Dot cols
E149    CB11        CL9:    RL      C                   ; Dot from pattern
E14B    34                  INC     (HL)                ;
E14C    35                  DEC     (HL)                ; Check CBUFF clear
E14D    200D                JR      NZ,CL12             ; NZ=Sprite above
E14F    7A                  LD      A,D                 ; A=Colours
E150    3004                JR      NC,CL10             ; NC=0 Pixel
E152    0F                  RRCA                        ;
E153    0F                  RRCA                        ;
E154    0F                  RRCA                        ;
E155    0F                  RRCA                        ; Select 1 colour
E156    E60F        CL10:   AND     0FH                 ;
E158    2001                JR      NZ,CL11             ; Z=Transparent
E15A    7B                  LD      A,E                 ; Use background
E15B    77          CL11:   LD      (HL),A              ; Colour in CBUFF
E15C    23          CL12:   INC     HL                  ;
E15D    10EA                DJNZ    CL9                 ; Next dot col
E15F    C1                  POP     BC                  ;
E160    10BE                DJNZ    CL6                 ; Next dot row
E162    E1                  POP     HL                  ;
E163    FDE1                POP     IY                  ;
E165    E1                  POP     HL                  ;
E166    D1                  POP     DE                  ;
E167    C1                  POP     BC                  ;
E168    C9                  RET                         ;

E169    78          SPRTES: LD      A,B                 ; A=Column
E16A    07                  RLCA                        ;
E16B    07                  RLCA                        ;
E16C    07                  RLCA                        ; A=X coord
E16D    C607                ADD     A,7                 ; RH edge of cell
E16F    47                  LD      B,A                 ; B=X coord
E170    79                  LD      A,C                 ; A=Row
E171    07                  RLCA                        ;
E172    07                  RLCA                        ;
E173    07                  RLCA                        ; A=Y coord
```

```
E174    C607              ADD     A,7             ; Bottom of cell
E176    4F                LD      C,A             ; C=Y coord
E177    AF                XOR     A               ; Sprite number
E178    CD8700      SS1:  CALL    CALATR          ; HL->Attributes
E17B    57                LD      D,A             ; D=Sprite number
E17C    CD4A00            CALL    RDVRM           ; Get Sprite Y
E17F    FED0              CP      208             ; Terminator?
E181    C8                RET     Z               ;
E182    D5                PUSH    DE              ;
E183    C5                PUSH    BC              ;
E184    CD8FE1            CALL    SPRITE          ; Do a sprite
E187    C1                POP     BC              ;
E188    F1                POP     AF              ;
E189    3C                INC     A               ; Next sprite number
E18A    FE20              CP      32              ; Done all?
E18C    20EA              JR      NZ,SS1          ;
E18E    C9                RET                     ;

E18F    91          SPRITE: SUB   C               ; (SY-Y)
E190    2F                CPL                     ; Make (Y-SY)
E191    FE27              CP      39              ; Possible overlap?
E193    D0                RET     NC              ;
E194    4F                LD      C,A             ; C=(Y-SY)
E195    23                INC     HL              ;
E196    CD4A00            CALL    RDVRM           ; Get Sprite X
E199    5F                LD      E,A             ;
E19A    78                LD      A,B             ; A=X coord
E19B    93                SUB     E               ;
E19C    5F                LD      E,A             ; E=(X-SX)
E19D    9F                SBC     A,A             ; Make 16 bit
E19E    57                LD      D,A             ; DE=(X-SX)
E19F    23                INC     HL              ;
E1A0    CD4A00            CALL    RDVRM           ; Get pattern#
E1A3    47                LD      B,A             ;
E1A4    23                INC     HL              ;
E1A5    CD4A00            CALL    RDVRM           ; Get EC & Colour
E1A8    CB7F              BIT     7,A             ; Early clock?
E1AA    2805              JR      Z,SP1           ;
E1AC    212000            LD      HL,32           ;
E1AF    19                ADD     HL,DE           ; Increase (X-SX)
E1B0    EB                EX      DE,HL           ;
E1B1    14          SP1:  INC     D               ;
E1B2    15                DEC     D               ; (X-SX)>255 or neg?
E1B3    C0                RET     NZ              ; NZ-Outside cell
E1B4    E60F              AND     0FH             ; Colour
E1B6    C8                RET     Z               ; Z=Transparent
E1B7    57                LD      D,A             ; D=Colour
E1B8    3AE0F3            LD      A,(RG1SAV)      ; Flags
E1BB    DB4F              BIT     1,A             ; SIZE
E1BD    0F                RRCA                    ; MAG
E1BE    3E08              LD      A,8             ; Minimum size
E1C0    3001              JR      NC,SP2          ;
E1C2    87                ADD     A,A             ; Double for MAG
```

```
E1C3    2800        SP2:    JR      Z,SP3           ;
E1C5    CB80                RES     0,B             ; Change pattern#
E1C7    CB88                RES     1,B             ;
E1C9    87                  ADD     A,A             ; Double for SIZE
E1CA    6F          SP3:    LD      L,A             ; L=Sprite size
E1CB    C606                ADD     A,6             ; Allow cell size
E1CD    B9                  CP      C               ;
E1CE    D8                  RET     C               ; Sprite above
E1CF    BB                  CP      E               ;
E1D0    D8                  RET     C               ; Sprite to left
E1D1    79                  LD      A,C             ;
E1D2    D607                SUB     7               ; (Y-SY) from top
E1D4    4F                  LD      C,A             ;
E1D5    7D                  LD      A,L             ; A=Sprite size
E1D6    2608                LD      H,8             ; Max dot rows
E1D8    3800                JR      C,SP5           ; C=Below cell top
E1DA    91                  SUB     C               ; A=Dot row overlap
E1DB    FE09                CP      9               ;
E1DD    3802                JR      C,SP4           ;
E1DF    3E08                LD      A,8             ;
E1E1    67          SP4:    LD      H,A             ; H=Row overlap
E1E2    7B          SP5:    LD      A,E             ;
E1E3    D607                SUB     7               ; (X-SX) from cell LH
E1E5    5F                  LD      E,A             ;
E1E6    7D                  LD      A,L             ; A=Sprite size
E1E7    2E08                LD      L,8             ; Max dot cols
E1E9    3808                JR      C,SP7           ; C=Past cell LH
E1EB    93                  SUB     E               ; A=Dot col overlap
E1EC    FE09                CP      9               ;
E1EE    3802                JR      C,SP6           ;
E1F0    3E08                LD      A,8             ;
E1F2    6F          SP6:    LD      L,A             ; L=Col overlap
E1F3    FD2151E2    SP7:    LD      IY,CBUFF        ; Results
E1F7    D5          SP8:    PUSH    DE              ;
E1F8    CB79                BIT     7,C             ; Reached sprite?
E1FA    2048                JR      NZ,SP15         ;
E1FC    E5                  PUSH    HL              ;
E1FD    FDE5                PUSH    IY              ;
E1FF    CB7B        SP9:    BIT     7,E             ; Reached sprite?
E201    2038                JR      NZ,SP14         ;
E203    FD7E00              LD      A,(IY+0)        ; CBUFF
E206    B7                  OR      A               ; Transparent?
E207    2032                JR      NZ,SP14         ;
E209    C5                  PUSH    BC              ;
E20A    D5                  PUSH    DE              ;
E20B    E5                  PUSH    HL              ;
E20C    3AE0F3              LD      A,(RG1SAV)      ; Flags
E10F    0F                  RRCA                    ; MAG
E210    3004                JR      NC,SP10         ;
E212    CB39                SRL     C               ; (Y-SY)/2
E214    CB3B                SRL     E               ; (X-SX)/2
E216    CB5B        SP10:   BIT     3,E             ; (X-SX)>7?
E218    2804                JR      Z,SP11          ;
```

```
E21A    CB9B            RES     3,E             ; (X-SX)-8
E21C    CBE1            SET     4,C             ; (Y-SY)+16
E21E    68      SP11:   LD      L,B             ;
E21F    2600            LD      H,0             ; HL=Pattern#
E221    44              LD      B,H             ; BC=Y offset
E222    29              ADD     HL,HL           ;
E223    29              ADD     HL,HL           ;
E224    29              ADD     HL,HL           ; HL=Pattern*8
E225    09              ADD     HL,BC           ; Select dot row
E226    ED4B26F9        LD      BC,(PATBAS)     ;
E22A    09              ADD     HL,BC           ; HL->Pattern
E22B    CD4A00          CALL    RDVRM           ; Get dot row
E22E    1C              INC     E               ;
E22F    07      SP12:   RLCA                    ; Select dot col
E230    1D              DEC     E               ;
E231    20FC            JR      NZ,SP12         ;
E233    3003            JR      NC,SP13         ; NC=0 Pixel
E235    FD7200          LD      (IY+0),D        ; Colour in CBUFF
E238    E1      SP13:   POP     HL              ;
E239    D1              POP     DE              ;
E23A    C1              POP     BC              ;
E23B    FD23    SP14:   INC     IY              ;
E23D    1C              INC     E               ; Right a dot col
E23E    2D              DEC     L               ; Finished cols?
E23F    20BE            JR      NZ,SP9          ;
E241    FDE1            POP     IY              ;
E243    E1              POP     HL              ;
E244    110800  SP15:   LD      DE,8            ;
E247    FD19            ADD     IY,DE           ;
E249    D1              POP     DE              ;
E24A    0C              INC     C               ; Down a dot row
E24B    25              DEC     H               ; Finished?
E24C    20A9            JR      NZ,SP8          ;
E24E    C9              RET                     ;

E24F    0000    BRKSTK: DEFW    0               ; Break stack

                ; ****************************
                ; * This buffer holds the 64 *
                ; * colour codes produced by *
                ; *       a cell scan:       *
                ; *                          *
                ; *   CCCCCCCC Bytes 00-07   *
                ; *   CCCCCCCC Bytes 08-15   *
                ; *   CCCCCCCC Bytes 16-23   *
                ; *   CCCCCCCC Bytes 24-31   *
                ; *   CCCCCCCC Bytes 32-39   *
                ; *   CCCCCCCC Bytes 40-47   *
                ; *   CCCCCCCC Bytes 48-55   *
                ; *   CCCCCCCC Bytes 56-64   *
                ; *                          *
                ; ****************************
```

```
E251                CBUFF:  DEFS    64                  ; Cell buffer

                            END
```

Character Editor

This program allows the MSX character patterns to be modified. When the program is first entered it copies the 2KB character set from its present location (usually the MSX ROM) to the CHRTAB buffer (E2A3H to EAA2H) and sets up the screen as shown below:

The program has two levels of operation, command and edit, with the RETURN key being used to toggle between them. In command mode the four arrow keys are used to select the character for editing. This is marked by a large cursor an is also displayed in magnified form on the right hand side of the screen. The "Q" key will quit the program and return to BASIC. The "A" key is used to adopt the character set, that is, to make it the system character set. When the character set is adopted it is copied to the highest part of memory (EB80H to F37FH) and its Slot ID and address placed in CGPNT.

In edit mode the four arrow keys are used to select the dot for editing, this is marked by a small cursor. The SPACE key will erase the current dot and the "." key set it. As the patter is modified the character menu on the left hand side of the screen is updated.

The character set in the CHRTAB may be saved on the cassette using a "BSAVE" statement and later re-loaded with a "BLOAD" statement. The ADOPT subroutine should be saved with the patterns and executed upon re-loading so that the system adopts the new character set. Alternatively the character set alone can be saved and its Slot ID and address placed in CGPNT upon re-loading using BASIC statements. Note that altering the character patterns does not affect the operation of the MSX system un the slightest.

```
                    ORG     0E000H
                    LOAD    0E000H

                ; ******************************
                ; *   BIOS STANDARD ROUTINES   *
                ; ******************************

                RDSLT:  EQU     000CH
                RDVRM:  EQU     004AH
                WRTVRM: EQU     004AH
                FILVRM: EQU     0056H
                INIGRP: EQU     0072H
                CHSNS:  EQU     009CH
                CHGET:  EQU     009FH
                MAPXYC: EQU     0111H
                FETCHC: EQU     0114H
                RSLREG: EQU     0138H

                ; ******************************
```

```
                    ; *     WORKSPACE VARIABLES    *
                    ; ****************************

                    GRPCOL: EQU     0F3D9H
                    FORCLR: EQU     0F3E9H
                    BAKCLR: EQU     0F3EAH
                    CGPNT:  EQU     0F91FH
                    EXPTBL: EQU     0FCC1H
                    SLTTBL: EQU     0FCC5H

                    ; ****************************
                    ; *      CONTROL CHARACTERS    *
                    ; ****************************

                    CR:     EQU     13
                    RIGHT:  EQU     28
                    LEFT:   EQU     29
                    UP:     EQU     30
                    DOWN:   EQU     31

E000    CDF6E0      CHEDIT: CALL    INIT            ; Cold start
E003    CDBDE0      CH1:    CALL    CHRMAG          ; Magnify chr
E006    CDFEE1              CALL    CHRXY           ; Chr coords
E009    1608                LD      D,8             ; Cursor size
E00B    CD2FE2              CALL    GETKEY          ; Get command
E00E    FE51                CP      "Q"             ; Quit
E010    C8                  RET     Z               ;
E011    2103E0              LD      HL,CH1          ; Set up return
E014    E5                  PUSH    HL              ;
E015    FE41                CP      "A"             ; Adopt
E017    CA6EE2              JP      Z,ADOPT         ;
E01A    FE0D                CP      CR              ; Edit
E01C    281F                JR      Z,EDIT          ;
E01E    0E01                LD      C,1             ; C=Offset
E020    FE1C                CP      RIGHT           ; Right
E022    2811                JR      Z,CH2           ;
E024    0EFF                LD      C,0FFH          ;
E026    FE1D                CP      LEFT            ; Left
E028    280B                JR      Z,CH2           ;
E02A    0EF0                LD      C,0F0H          ;
E02C    FE1E                CP      UP              ; Up
E02E    2805                JR      Z,CH2           ;
E030    0E10                LD      C,16            ;
E032    FE1F                CP      DOWN            ; Down
E034    C0                  RET     NZ              ;
E035    3AA1E2      CH2:    LD      A,(CHRNUM)      ; Current chr
E038    81                  ADD     A,C             ; Add offset
E039    32A1E2              LD      (CHRNUM),A      ; New chr
E03C    C9                  RET                     ;

E03D    CDE6E1      EDIT:   CALL    DOTXY           ; Dot coords
E040    1602                LD      D,2             ; Cursor size
E042    CD2FE2              CALL    GETKEY          ; Get command
```

```
E045    FE0D            CP      CR              ; Quit
E047    C8              RET     Z               ;
E048    213DE0          LD      HL,EDIT         ; Set up return
E04B    E5              PUSH    HL              ;
E04C    0100FE          LD      BC,0FE00H       ; AND/OR masks
E04F    FE20            CP      " "             ; Space
E051    2824            JR      Z,ED3           ;
E053    0C              INC     C               ; New OR mask
E054    FE2E            CP      "."             ; Dot
E056    281F            JR      Z,ED3           ;
E058    FE1C            CP      RIGHT           ; Right
E05A    2811            JR      Z,ED2           ;
E05C    0EFF            LD      C,0FFH          ; C=Offset
E05E    FE1D            CP      LEFT            ; Left
E060    280B            JR      Z,ED2           ;
E062    0EF8            LD      C,0F8H          ;
E064    FE1E            CP      UP              ; Up
E066    2805            JR      Z,ED2           ;
E068    0E08            LD      C,8             ;
E06A    FE1F            CP      DOWN            ; Down
E06C    C0              RET     NZ              ;
E06D    3AA2E2  ED2:    LD      A,(DOTNUM)      ; Current dot
E070    81              ADD     A,C             ; Add offset
E071    E63F            AND     63              ; Wrap round
E073    32A2E2          LD      (DOTNUM),A      ; New dot
E076    C9              RET                     ;
E077    CD1EE2  ED3:    CALL    PATPOS          ; IY->Pattern
E07A    3AA2E2          LD      A,(DOTNUM)      ; Current dot
E07D    F5              PUSH    AF              ;
E07E    0F              RRCA                    ;
E07F    0F              RRCA                    ;
E080    0F              RRCA                    ;
E081    E607            AND     7               ; A=Row
E083    5F              LD      E,A             ;
E084    1600            LD      D,0             ; DE=Row
E086    FD19            ADD     IY,DE           ; IY->Row
E088    F1              POP     AF              ;
E089    E607            AND     7               ; A=Column
E08B    3C              INC     A               ;
E08C    CB08    ED4:    RRC     B               ; AND mask
E08E    CB09            RRC     C               ; OR mask
E090    3D              DEC     A               ; Count columns
E091    20F9            JR      NZ,ED4          ;
E093    FD7E00          LD      A,(IY+0)        ; A=Pattern
E096    A0              AND     B               ; Strip old bit
E097    B1              OR      C               ; New bit
E098    FD7700          LD      (IY+0),A        ; New pattern
E09B    CDBDE0          CALL    CHRMAG          ; Update magnified

E09E    CD1EE2  CHROUT: CALL    PATPOS          ; IY->Pattern
E0A1    CDFEE1          CALL    CHRXY           ; Get coords
E0A4    CDA3E1          CALL    MAP             ; Map
E0A7    0608            LD      B,8             ; Dot rows
```

```
E0A9    D5          CO1:    PUSH    DE              ;
E0AA    E5                  PUSH    HL              ;
E0AB    3E08                LD      A,8             ; Dot cols
E0AD    FD5E00              LD      E,(IY+0)        ; E=Pattern
E0B0    CDC4E1              CALL    SETROW          ; Set row
E0B3    E1                  POP     HL              ; HL=CLOC
E0B4    D1                  POP     DE              ; D=CMASK
E0B5    CDB8E1              CALL    DOWNP           ; Down a pixel
E0B8    FD23                INC     IY              ;
E0BA    10ED                DJNZ    CO1             ;
E0BC    C9                  RET                     ;

E0BD    CD1EE2      CHRMAG: CALL    PATPOS          ; IY->Pattern
E0C0    0EBF                LD      C,191           ; Start X
E0C2    1E07                LD      E,7             ; Start Y
E0C4    CDA3E1              CALL    MAP             ; Map
E0C7    0608                LD      B,8             ; Dot rows
E0C9    0E05        CM1:    LD      C,5             ; Row mag
E0CB    C5          CM2:    PUSH    BC              ;
E0CC    D5                  PUSH    DE              ;
E0CD    E5                  PUSH    HL              ;
E0CE    0608                LD      B,8             ; Dot columns
E0D0    FD7E00              LD      A,(IY+0)        ; A=Pattern
E0D3    07          CM3:    RLCA                    ; Test bit
E0D4    F5                  PUSH    AF              ;
E0D5    9F                  SBC     A,A             ; 0=00H, 1=FFH
E0D6    5F                  LD      E,A             ; E=Mag pattern
E0D7    3E05                LD      A,5             ; Column mag
E0D9    CDC4E1              CALL    SETROW          ; Set row
E0DC    CDAEE1              CALL    RIGHTP          ; Right a pixel
E0DF    CDAEE1              CALL    RIGHTP          ; Skip grid
E0E2    F1                  POP     AF              ;
E0E3    10EE                DJNZ    CM3             ;
E0E5    E1                  POP     HL              ; HL=CLOC
E0E6    D1                  POP     DE              ; D=CMASK
E0E7    C1                  POP     BC              ;
E0E8    CDB8E1              CALL    DOWNP           ; Down a pixel
E0EB    0D                  DEC     C               ;
E0EC    20DD                JR      NZ,CM2          ;
E0EE    CDB8E1              CALL    DOWNP           ; Skip grid
E0F1    FD23                INC     IY              ;
E0F3    10D4                DJNZ    CM1             ;
E0F5    C9                  RET                     ;

E0F6    100008      INIT:   LD      BC,2048         ; Size
E0F9    11A3E2              LD      DE,CHRTAB       ; Destination
E0FC    2A20F9              LD      HL,(CGPNT+1)    ; Source
E0FF    C5          IN1:    PUSH    BC              ;
E100    D5                  PUSH    DE              ;
E101    3A1FF9              LD      A,(CGPNT)       ; Slot ID
E104    CD0C00              CALL    RDSLT           ; Read chr  pattern
E107    FB                  EI                      ;
E108    D1                  POP     DE              ;
```

```
E109    C1                  POP     BC                  ;
E10A    12                  LD      (DE),A              ; Put in buffer
E10B    13                  INC     DE                  ;
E10C    23                  INC     HL                  ;
E10D    0B                  DEC     BC                  ;
E10E    78                  LD      A,B                 ;
E10F    B1                  OR      C                   ;
E110    20ED                JR      NZ,IN1              ;
E112    CD7200              CALL    INIGRP              ; SCREEN 2
E115    3AE9F3              LD      A,(FORCLR)          ; Colour 1
E118    07                  RLCA                        ;
E119    07                  RLCA                        ;
E11A    07                  RLCA                        ;
E11B    07                  RLCA                        ;
E11C    4F                  LD      C,A                 ; C=Colour 1
E11D    3AEAF3              LD      A,(BAKCLR)          ; Colour 0
E120    B1                  OR      C                   ; Mix
E121    010018              LD      BC,6144             ; Colour table size
E124    2AC9F3              LD      HL,(GRPCOL)         ; Colour table
E127    CD5600              CALL    FILVRM              ; Fill colours
E12A    210BB1              LD      HL,177*256+11       ;
E12D    010AFF              LD      BC,0FFH*256+10      ;
E130    1E06                LD      E,6                 ;
E132    3E11                LD      A,17                ;
E134    CD62E1              CALL    GRID                ; Draw chr grid
E137    210631              LD      HL,49*256+6         ;
E13A    01BEAA              LD      BC,0AAH*256+190     ;
E13D    1E06                LD      E,6                 ;
E13F    3E09                LD      A,9                 ;
E141    CD62E1              CALL    GRID                ; Draw mag grid
E144    213031              LD      HL,49*256+48        ;
E147    01BEFF              LD      BC,0FFH*256+190     ;
E14A    1E06                LD      E,6                 ;
E14C    3E02                LD      A,2                 ;
E14E    CD62E1              CALL    GRID                ; Draw mag box
E151    AF                  XOR     A                   ;
E152    32A2E2              LD      (DOTNUM),A          ; Current dot
E155    21A1E2              LD      HL,CHRNUM           ;
E158    77                  LD      (HL),A              ; Current chr
E159    E5          IN2:    PUSH    HL                  ;
E15A    CD9EE0              CALL    CHROUT              ; Display chr
E15D    E1                  POP     HL                  ;
E15E    34                  INC     (HL)                ; Next chr
E15F    20F8                JR      NZ,IN2              ; Do 256
E161    C9                  RET                         ;

E162    F5          GRID:   PUSH    AF                  ;
E163    C5                  PUSH    BC                  ;
E164    E5                  PUSH    HL                  ;
E165    CDA3E1              CALL    MAP                 ; Map
E168    C1                  POP     BC                  ; B=Len,C=Step
E169    F1                  POP     AF                  ;
E16A    5F                  LD      E,A                 ; E=Pattern
```

```
E16B    F1              POP     AF              ; A=Count
E16C    F5              PUSH    AF              ;
E16D    D5              PUSH    DE              ;
E16E    E5              PUSH    HL              ;
E16F    F5      GR1:    PUSH    AF              ;
E170    C5              PUSH    BC              ;
E171    D5              PUSH    DE              ;
E172    E5              PUSH    HL              ;
E173    78              LD      A,B             ; A=Len
E174    CDC4E1          CALL    SETROW          ; Horizontal line
E177    E1              POP     HL              ; HL=CLOC
E178    D1              POP     DE              ; D=CMASK
E179    CDB8E1  GR3:    CALL    DOWNP           ; Down a pixel
E17C    0D              DEC     C               ; Done step?
E17D    20FA            JR      NZ,GR3          ;
E17F    C1              POP     BC              ;
E180    F1              POP     AF              ; A=Count
E181    3D              DEC     A               ; Done lines?
E182    20EB            JR      NZ,GR1          ;
E184    E1              POP     HL              ; HL=Initial CLOC
E185    D1              POP     DE              ; D=Initial CMASK
E186    F1              POP     AF              ; A=Count
E187    F5      GR4:    PUSH    AF              ;
E188    C5              PUSH    BC              ;
E189    D5              PUSH    DE              ;
E18A    E5              PUSH    HL              ;
E18B    3E01    GR5:    LD      A,1             ; Line width
E18D    CDC4E1          CALL    SETROW          ; Thin line
E190    CDB8E1          CALL    DOWNP           ; Down a pixel
E193    10F6            DJNZ    GR5             ; Vertical len
E195    E1              POP     HL              ; HL=CLOC
E196    D1              POP     DE              ; D=CMASK
E197    CDAEE1  GR6:    CALL    RIGHTP          ; Right a pixel
E19A    0D              DEC     C               ; Done step?
E19B    20FA            JR      NZ,GR6          ;
E19D    C1              POP     BC              ;
E19E    F1              POP     AF              ; A=Count
E19F    3D              DEC     A               ; Done lines?
E1A0    20E5            JR      NZ,GR4          ;
E1A2    C9              RET                     ;

E1A3    0600    MAP:    LD      B,0             ; X MSB
E1A5    50              LD      D,B             ; Y MSB
E1A6    CD1101          CALL    MAPXYC          ; Map coords
E1A9    CD1401          CALL    FETCHC          ; HL=CLOC
E1AC    57              LD      D,A             ; D=CMASK
E1AD    C9              RET                     ;

E1AE    CB0A    RIGHTP: RRC     D               ; Shift CMASK
E1B0    D0              RET     NC              ; NC=Same cell
E1B1    C5      RP1:    PUSH    BC              ;
E1B2    010800          LD      BC,8            ; Offset
E1B5    09              ADD     HL,BC           ; HL=Next cell
```

```
E1B6    C1                  POP     BC                  ;
E1B7    C9                  RET                         ;

E1B8    23          DOWNP:  INC     HL                  ; CLOC down
E1B9    7D                  LD      A,L                 ;
E1BA    E607                AND     7                   ; Select pixel row
E1BC    C0                  RET     NZ                  ; NZ=Same cell
E1BD    C5                  PUSH    BC                  ;
E1BE    01F800              LD      BC,00F8H            ; Offset
E1C1    09                  ADD     HL,BC               ; HL=Next cell
E1C2    C1                  POP     BC                  ;
E1C3    C9                  RET                         ;

E1C4    C5          SETROW: PUSH    BC                  ;
E1C5    47                  LD      B,A                 ; B=Count
E1C6    CD4A00      SE1:    CALL    RDVRM               ; Get old pattern
E1C9    4F          SE2:    LD      C,A                 ; C=Old
E1CA    7A                  LD      A,D                 ; A=CMASK
E1CB    2F                  CPL                         ; AND mask
E1CC    A1                  AND     C                   ; Strip old bit
E1CD    CB03                RLC     E                   ; Shift pattern
E1CF    3001                JR      NC,SE3              ; NC=0 Pixel
E1D1    B2                  OR      D                   ; Set 1 Pixel
E1D2    05          SE3:    DEC     B                   ; Finished?
E1D3    280C                JR      Z,SE4               ;
E1D5    CB0A                RRC     D                   ; CMASK right
E1D7    30F0                JR      NC,SE2              ; NC=Same cell
E1D9    CD4D00              CALL    WRTVRM              ; Update cell
E1DC    CDB1E1              CALL    RP1                 ; Next cell
E1DF    18E5                JR      SE1                 ; Start again
E1E1    CD4D00      SE4:    CALL    WRTVRM              ; Update cell
E1E4    C1                  POP     BC                  ;
E1E5    C9                  RET                         ;

E1E6    3AA2E2      DOTXY:  LD      A,(DOTNUM)          ; Current dot
E1E9    F5                  PUSH    AF                  ;
E1EA    E607                AND     7                   ; Column
E1EC    07                  RLCA                        ;
E1ED    4F                  LD      C,A                 ; C=Col*2
E1EE    07                  RLCA                        ; A=Col*4
E1EF    81                  ADD     A,C                 ; A=Col*6
E1F0    C6BF                ADD     A,191               ; Grid atart
E1F2    4F                  LD      C,A                 ; C=X coord
E1F3    F1                  POP     AF                  ;
E1F4    E638                AND     38H                 ; Row*8
E1F6    0F                  RRCA                        ;
E1F7    5F                  LD      E,A                 ; E=Row*4
E1F8    0F                  RRCA                        ; A=Row*2
E1F9    83                  ADD     A,E                 ; A=Row*6
E1FA    C607                ADD     A,7                 ; Grid start
E1FC    5F                  LD      E,A                 ; E=Y coord
E1FD    C9                  RET                         ;
```

```
E1FE    3AA1E2      CHRXY:  LD      A,(CHRNUM)      ; Current chr
E201    F5                  PUSH    AF              ;
E202    CD14E2              CALL    MULT11          ; Column*11
E205    C60C                ADD     A,12            ; Grid start
E207    4F                  LD      C,A             ; C=X coord
E208    F1                  POP     AF              ;
E209    0F                  RRCA                    ;
E20A    0F                  RRCA                    ;
E20B    0F                  RRCA                    ;
E20C    0F                  RRCA                    ;
E20D    CD14E2              CALL    MULT11          ; Row*11
E210    C608                ADD     A,8             ; Grid start
E212    5F                  LD      E,A             ; E=Y coord
E213    C9                  RET                     ;

E214    E60F        MULT11: AND     0FH             ;
EF16    57                  LD      D,A             ; D=N
E217    07                  RLCA                    ;
E218    47                  LD      B,A             ; B=N*2
E219    07                  RLCA                    ;
E21A    07                  RLCA                    ; A=N*8
E21B    80                  ADD     A,B             ;
E21C    82                  ADD     A,D             ; A=N*11
E21D    C9                  RET                     ;

E21E    3AA1E2      PATPOS: LD      A,(CHRNUM)      ; Current chr
E221    6F                  LD      L,A             ;
E222    2600                LD      H,0             ; HL=Chr
E224    29                  ADD     HL,HL           ;
E225    29                  ADD     HL,HL           ;
E226    29                  ADD     HL,HL           ; HL=Chr*8
E227    EB                  EX      DE,HL           ; DE=Chr*8
E228    FD21A3E2            LD      IY,CHRTAB       ; Patterns
E22C    FD19                ADD     IY,DE           ; IY->Pattern
E22E    C9                  RET                     ;

E22F    0600        GETKEY: LD      B,0             ; Cursor flag
E231    C5          GE1:    PUSH    BC              ; C=X coord
E232    D5                  PUSH    DE              ; E=Y coord
E233    CD50E2              CALL    INVERT          ; Flip cursor
E236    D1                  POP     DE              ;
E237    C1                  POP     BC              ;
E238    04                  INC     B               ; Flip flag
E239    21401F              LD      HL,8000         ; Blink rate
E23C    CD9C00      GE2:    CALL    CHSNS           ; Check KEYBUF
E23F    2007                JR      NZ,GE3          ; NZ=Got key
E241    2B                  DEC     HL              ;
E242    7C                  LD      A,H             ;
E243    B5                  OR      L               ;
E244    20F6                JR      NZ,GE2          ;
E246    18E9                JR      GE1             ; Time for cursor
E248    CB40        GE3:    BIT     0,B             ; Cursor state
E24A    C450E2              CALL    NZ,INVERT       ; Remove cursor
```

```
E24D    C39F00              JP      CHGET               ; Collect character

E250    D5          INVERT: PUSH    DE                  ;
E251    CDA3E1              CALL    MAP                 ; Map coords
E254    F1                  POP     AF                  ; A=Cursor size
E255    47                  LD      B,A                 ; B=Rows
E256    5F                  LD      E,A                 ; E=Cols
E257    D5          IV1:    PUSH    DE                  ;
E258    E5                  PUSH    HL                  ;
E259    CD4A00      IV2:    CALL    RDVRM               ; Old pattern
E25C    AA                  XOR     D                   ; Flip a bit
E25D    CD4D00              CALL    WRTVGM              ; Put it back
E260    CDAEE1              CALL    RIGHTP              ; Right a pixel
E263    1D                  DEC     E                   ;
E264    20F3                JR      NZ,IV2              ;
E266    E1                  POP     HL                  ; HL=CLOC
E267    D1                  POP     DE                  ; D=CMASK
E268    CDB8E1              CALL    DOWNP               ; Down a pixel
E26B    10EA                DJNZ    IV1                 ;
E26D    C9                  RET                         ;

E26E    010008      ADOPT:  LD      BC,2048             ; Size
E271    1180EB              LD      DE,0EB80H           ; Destination
E274    ED5320F9            LD      (CGPNT+1),DE        ;
E278    21A3E2              LD      HL,CHRTAB           ; Source
E27B    EDB0                LDIR                        ; Copy up high
E27D    CD3801              CALL    RSLREG              ; Read PSLOT reg
E280    07                  RLCA                        ;
E281    07                  RLCA                        ;
E282    E603                AND     3                   ; Select Page 3
E284    4F                  LD      C,A                 ;
E285    0600                LD      B,0                 ; BC=Page 3 PSLOT#
E287    21C1FC              LD      HL,EXPTBL           ; Expanders
E28A    09                  ADD     HL,BC               ;
E28B    CB7E                BIT     7,(HL)              ; PSLOT expanded?
E28D    280E                JR      Z,AD1               ; A=Normal
E28F    21C5FC              LD      HL,SLTTBL           ; Secondary regs
E292    09                  ADD     HL,BC               ;
E293    7E                  LD      A,(HL)              ; A=Secondary reg
E294    07                  RLCA                        ;
E295    07                  RLCA                        ;
E296    07                  RLCA                        ;
E297    07                  RLCA                        ;
E298    E60C                AND     0CH                 ; A=Page 3 SSLOT#
E29A    B1                  OR      C                   ; Mix Page 3 PSLOT#
E29B    CBFF                SET     7,A                 ; A=Slot ID
E29D    321FF9      AD1:    LD      (CGPNT),A           ;
E2A0    C9                  RET                         ;

E2A1    00          CHRNUM: DEFB    0                   ; Current chr
E2A2    00          DOTNUM: DEFB    0                   ; Current dot
E2A3                CHRTAB: DEFS    2048                ; Patterns to EAA2H
```

```
END
```