15 The Main Firmware Jumpblock.

This section describes in detail the entry and exit conditions and the effects of all the routines in the main firmware jumpblock. The main firmware jumpblock is described in section 14.1.

The user is advised to read the sections on each pack before attempting to understand the jumpblock entries. The relevant sections are:

Key Manager	(KM)	Section 3.
Text VDU	(TXT)	Section 4.
Graphics VDU	(GRA)	Section 5.
Screen Pack	(SCR)	Section 6.
Sound Manager	(SOUND)	Section 7.
Cassette Manager	(CAS)	Section 8.
AMSDOS		Section 9.
Kernel	(KL)	Section 2, 10,11 and 12.
Maching Pack	(MC)	Section 13.

The top line of each description has the following layout:

<Entry number>: <Entry name> <Entry address>

Entries in the jumpblock are numbered starting from zero. The entry address is the address to call to invoke the firmware routine or the address of the three bytes to patch to intercept the routine. The entry address can be calculated as:

Entry address = Start of jumpblock +3 * Entry number

Each entry is named and is referred to by name throughout this manual.

The last section of each description is a list of related routines. The user is advised to look at these as the list may include routines more suited for the application being considered. Conversely the routines may shed further light on how the original routines should be used.

The descriptions of the routines are for the default routine that the entry jumps to. The user may change the entry and this may alter the action of the routine. The user is advised to stick to the entry/exit conditions described otherwise programs that call the routine (BASIC for example) may cease to operate correctly.

0: KM INITIALISE #BB00

Initialize the Key Manager

Action:

Full initialization of the Key Manager (as during EMS). All Key Manager variables, buffers and indirections are initialized. The previous state of the Key Manager is lost.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The Key Manager indirection (KM TEST KEY) is set to its default routine.

The key buffer is set up (to be empty).

The expansion buffer is set up and the expansions are set to their default strings.

The translation table are initialized to their default translations.

The repeating key map is initialized to its default state.

The repeat speeds are set to their default values.

Shift and caps lock are turned off.

The break event is disarmed.

See Appendices II, III and IV for the default translation tables, repeating key table and expansion strings.

This routine enables interrupts.

Related entries:

KM RESET

1: KM RESET #BB03

Reset the Key Manager.

Action:

Reinitializes the Key Manager indirections and buffers.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The Key Manager indirection (KM TEST KEY) is set to its default routine.

The key buffer is set up (to be empty).

The expansion buffer is set up and the expansions are set to their default strings (see Appendix IV).

The break event is disarmed.

All pending keys and characters are discarded.

This routine enables interrupts.

Related entries:

KM DISARM BREAK
KM EXP BUFFER
KM INITIALISE

2: KM WAIT CHAR #BB06

Wait for the next character from the keyboard.

Action:

Try to get a character from the key buffer or the current expansion string. This routine waits until a character is available if no character is immediately available.

Entry conditions:

No conditions.

Exit conditions:

Carry true.

A contains the character.

Other flags corrupt.

All other registers preserved.

Notes:

The possible sources for generating the next character are, in the order that they are tested:

The 'put back' character.
The next character of an expansion string.
The first character of an expansion string.
A character from a key translation table.

Expansion tokens found in the key translation table are expanded to their associated strings. Expansion tokens found in expansion strings are not expanded but are treated as characters.

Related entries:

KM CHAR RETURN
KM READ CHAR
KM WAIT KEY

3: KM READ CHAR #BB09

Test if a character is available from the keyboard.

Action:

Try to get a character from the key buffer or the current expansion string. This routine does not wait for a character to become available if there is no character available immediately.

Entry conditions:

No conditions.

Exit conditions:

If there was a character available:

Carry true.
A contains the character.

If there was no character available:

Carry false.
A corrupt.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The possible sources for generating the next character are, in the order that they are tested:

The 'put back' character.
The next character of an expansion string.
The first character of an expansion string.
A character from a key translation table.

Expansion tokens in the key translation table will be expanded to their associated strings. Expansion tokens found in expansion strings are not expanded but are treated as characters.

This routine will always return a character if one is available. It is therefore possible to flush out the Key Manager buffers by calling KM READ CHAR repeatedly until it reports that no character is available.

Related entries:

KM CHAR RETURN
KM FLUSH
KM READ KEY
KM WAIT CHAR

4: KM CHAR RETURN #BB0C

Return a single character to the keyboard for next time.

Action:

Save a character for the next call of KM READ CHAR or KM WAIT CHAR.

Entry conditions:

A contains the character to put back.

Exit conditions:

All registers and flags preserved.

Notes:

The 'put back' character will be returned before any other character is generated by the keyboard. It will not be expanded (or otherwise dealt with) but will be returned as it is. The 'put back' character need not have been read from the keyboard, it could be inserted by the user for some purpose.

It is only possible to have one 'put back' character. If this routine is called twice without reading a character between these then the first ‘put back' will be lost. Furthermore, it is not possible to return character 255 (because this is used as the marker for no 'put back' character).

Related entries:

KM READ CHAR
KM WAIT CHAR

5: KM SET EXPAND #BB0F

Set an expansion string.

Action:

Set the expansion string associated with an expansion token.

Entry conditions:

B contains the expansion token for the expansion to set.

C contains the length of the string.

HL contains the address of the string.

Exit conditions:

If the expansion is OK:
    Carry true.

If the string was too long or the token was invalid:
    Carry false.

Always:
    A, BC, DE, HL and other flags corrupt.
    All other registers preserved.

Notes:

The string to be set may lie anywhere in RAM. Expansion strings cannot be set directly from ROM.

The characters in the string are not expanded (or otherwise dealt with). It is therefore possible to put any character into an expansion string.

If there is insufficient room in the expansion buffer for the new string then no change is made to the expansions.

If the string set is currently being used to generate characters (by KM READ CHAR or KM WAIT CHAR) then the unread portion of the string is discarded. The next character will be read from the key buffer.

This routine enables interrupts.

Related entries:

KM GET EXPAND
KM READ CHAR
KM WAIT CHAR

6: KM GET EXPAND #BB12

Get a character from an expansion string.

Action:

Read a character from an expansion string. The characters in the string are numbered starting from 0.

Entry conditions:

A contains an expansion token.

L contains the character number.

Exit conditions:

If the character was found:

Carry true.

A contains the character.

If the token was invalid or the string was not long enough:

Carry false.

A corrupt.

Always:

DE and other flags corrupt.
All other registers preserved.

Notes:

The characters in the expansion string are not expanded (or otherwise dealt with). It is therefore possible to put any character into an expansion string.

Related entries:

KM READ CHAR
KM SET EXPAND

7: KM EXP BUFFER #BB15

Allocate a buffer for expansion strings.

Action:

Set the address and length of the expansion buffer. Initialize the buffer with the default expansion strings.

Entry conditions:

DE contains the address of the buffer.

HL contains the length of the buffer.

Exit conditions:

If the buffer is OK:

  Carry true.

If the buffer is too short.

  Carry false.

Always:

  A, BC, DE, HL and other flags corrupt.
  All other registers preserved.

Notes:

The buffer must not be located underneath a ROM and it must be at least 49 bytes long (i.e. have sufficient space for the default expansion strings). If the new buffer is too short then the old buffer is left unchanged.

The default expansion strings are given in Appendix IV.

Any expansion string currently being read is discarded.

This routine enables interrupts.

Related entries:

KM GET EXPAND
KM SET EXPAND

8: KM WAIT KEY #BB18

Wait for next key from the keyboard.

Action:

Try to get a key from the key buffer. This routine waits until a key is found if no key is immediately available.

Entry conditions:

No conditions.

Exit conditions:

Carry true.
A contains the character or expansion token.

Other flags corrupt.
All registers preserved.

Notes:

The next key is read from the key buffer and translated using the appropriate key translation table. Expansion tokens are not expanded but are passed out for the user to deal with, as are normal characters. Other Key Manager tokens (shift lock, caps lock and ignore) are obeyed but are not passed out.

Related entries:

KM READ KEY
KM WAIT CHAR

9: KM READ KEY #BB1B

Test if a key is available from the keyboard.

Action:

Try to get a key from the key buffer. This routine does not wait if no key is available immediately.

Entry conditions:

No conditions.

Exit conditions:

If a key was available:

Carry true.

A contains the character or expansion token.

If no key was available:

Carry false.

A corrupt.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The next key is read from the key buffer and translated using the appropriate key translation table. Expansion tokens are not expanded but are passed out for the user to deal with, as are normal characters. Other Key Manager tokens (shift lock, caps lock and ignore) are obeyed but are not passed out.

This routine will always return a key if one is available. It is therefore possible to flush out the key buffer by calling KM READ KEY repeatedly until it claim no key is available. Note, however, that the 'put back' character or a partially read expansion string is ignored. It is advisable to use KM READ CHAR to flush these out when emptying the Key Manager buffers, or, in V1.1 firmware, to call KM FLUSH.

Related entries:

KM FLUSH
KM READ CHAR
KM WAIT KEY

10: KM TEST KEY #BB1E

Test if a key is pressed.

Action:

Test if a particular key or joystick button is pressed. This is done using the key state map rather then by accessing the keyboard hardware.

Entry conditions:

A contains the key number.

Exit conditions:

If the key is pressed:

Zero false.

If the key is not pressed:

Zero true.

Always:

Carry false.

C contains the current shift and control state.

A, HL and other flags corrupt.
All other registers preserved.

Notes:

The shift and control states are automatically read when a key is scanned. If bit 7 is set then the control key is pressed and if bit 5 is set then one of the shift keys is pressed.

The key number is not checked. An invalid key number will generate the correct shift and control states but the state of the key tested will be meaningless.

The key state map which this routine tests is updated by the keyboard scanning routine. Normally this run is every fiftieth of a second and so the state may be out of date by that much. The key debouncing requires that a key should be released for two scans of the keyboard before it is marked as released in the key state map; the pressing of a key is detected immediately.

Related entries:

KM GET JOYSTICK
KM GET STATE
KM READ KEY

11: KM GET STATE #BB21

Fetch Caps Lock and Shift Lock states.

Action:

Ask if the keyboard is currently shift locked or caps locked.

Entry conditions:

No conditions.

Exit conditions:

L contains the shift lock state.
H contains the caps lock state.

AF corrupt.
All other registers preserved.

Notes:

The lock states are:

#00 means the lock is off
#FF means the lock is on

The default lock states are off.

Related entries:

KM SET LOCKS
KM TEST KEY

12: KM GET JOYSTICK #BB24

Fetch current state of the joystick(s).

Action:

Ask what the current states of the joysticks are. These are read from the key state map rather than by accessing the keyboard hardware.

Entry conditions:

No conditions.

Exit conditions:

H contains the state of joystick 0.
L contains the state of joystick 1.
A contains the state of joystick 0.

Flags corrupt.
All other registers preserved.

Notes:

In normal operation the key state map is updated by the key scanning routine every fiftieth of a second so the state returned may be slightly out of date.

The joystick states are bit significant as follows:

Bit 0	Up.
Bit 1	Down.
Bit 2	Left.
Bit 3	Right.
Bit 4	Fire 2.
Bit 5	Fire 1.
Bit 6	Spare joystick button (usually unconnected).
Bit 7	Always zero.

If a bit is set then the appropriate button is pressed.

Joystick 1 is indistinguishable from certain keys on the keyboard (see Appendix I).

Related entries:

KM TEST KEY

13: KM SET TRANSLATE #BB27

Set entry in normal key translation table.

Action:

Set what character or token a key will be translated to when neither shift nor control is pressed.

Entry conditions:

A contains a key number.

B contains the new translation.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.

#FD is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).

#FE is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).

#FF is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET TRANSLATE
KM SET CONTROL
KM SET SHIFT

14: KM GET TRANSLATE #BB2A

Get entry from normal translation table.

Action:

Ask what character or token a key will be translated to when neither shift nor control is pressed.

Entry conditions:

A contains a key number

Exit conditions:

A contains the current translation.

HL and flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the translation returned is meaningless.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.

#FD is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).

#FE is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).

#FF is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET SHIFT
KM SET TRANSLATE

15: KM SET SHIFT #BB2D

Set entry in shifted key translation table.

Action:

Set what character or token a key will be translated to when control is not pressed but shift is pressed or shift lock is on.

Entry conditions:

A contains a key number.

B contains the new translation.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.

#FD is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).

#FE is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).

#FF is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET SHIFT
KM SET TRANSLATE

16: KM GET SHIFT #BB30

Get entry from shifted key translation table.

Action:

Ask what character or token a key will be translated to when control is not pressed but shift is pressed or shift lock is on.

Entry conditions:

A contains a key number.

Exit conditions:

A contains the current translation.

HL and flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the translation returned is meaningless.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET SHIFT
KM SET TRANSLATE

17: KM SET CONTROL #BB33

Set entry in control key translation table.

Action:

Set a character or token a key will be translated to when control is pressed.

Entry conditions:

A contains a key number.
B contains the new translation.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET SHIFT
KM SET TRANSLATE

18: KM GET CONTROL #BB36

Get entry from control key translation table.

Action:

Ask what a character or token a key will be translated to when control is pressed.

Entry conditions:

A contains a key number.

Exit conditions:

A contains the current translation.

HL and flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the translation returned is meaningless.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET SHIFT
KM SET TRANSLATE

19: KM SET REPEAT #BB39

Set whether a key may repeat.

Action:

Set the entry in the repeating key map that determines whether a key is allowed to repeat or not.

Entry conditions:

If the key is to be allowed to repeat:
    B contains #FF.

If the key is not to be allowed to repeat:
    B contains #00.

Always:
    A contains the key number.

Exit conditions:

AF, BC and HL corrupt.
All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

The default repeating keys are listed in Appendix III.

Related entries:

KM GET REPEAT
KM SET DELAY

20: KM GET REPEAT #BB3C

Ask if a key is allowed to repeat.

Action:

Test the entry in the repeating key map that says whether a key is allowed to repeat or not.

Entry conditions:

A contains a key number.

Exit conditions:

If the key is allowed to repeat:
    Zero false.

If the key is not allowed to repeat:
    Zero true.

Always:
    Carry false.

    A, HL and other flags corrupt.
    All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the repeat state returned is meaningless.

The default repeating keys are listed in Appendix III.

Related entries:

KM SET REPEAT

21: KM SET DELAY #BB3F

Set start delay and repeat speed.

Action:

Set the time before keys first repeat (start up delay) and the time between repeats (repeat speed).

Entry conditions:

H contains the new start up delay.

L contains the new repeat speed.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

Both delays are given in scans of the keyboard. The keyboard is scanned every fiftieth of a second.

A start up delay or repeat speed of 0 is taken to mean 256.

The default start up delay is 30 scans (0.6 seconds) and the default repeat speed is 2 scans (0.04 seconds or 25 characters a second).

Note that a key is prevented from repeating (by the key scanner) if the key buffer is not empty. Thus the actual repeat speed is the slower of the supplied repeat speed and the rate at which characters are removed from the buffer. This is intended to prevent the user from getting too far ahead of a program that is running sluggishly.

The start up delay and repeat speed apply to all keys on the keyboard that are set to repeat.

Related entries:

KM GET DELAY
KM SET REPEAT

22: KM GET DELAY #BB42

Get start up delay and repeat speed.

Action:

Ask the time before keys first repeat (start up delay) and the time between repeats (repeat speed).

Entry conditions:

No conditions.

Exit conditions:

H contains the start up delay.
L contains the repeat speed.

AF corrupt.
All other registers preserved.

Notes:

Both delays are given in scans of the keyboard. The keyboard is scanned every fiftieth of a second.

A repeat speed or start up delay of 0 means 256.

Related entries:

KM SET DELAY

23: KM ARM BREAK #BB45

Allow break events to be generated.

Action:

Arm the break mechanism. The next call of KM BREAK EVENT will generate a break event.

Entry conditions:

DE contains the address of the break event routine.

C contains the ROM select address for this routine.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The break mechanism can be disarmed by calling KM DISARM BREAK (or KM RESET).

This routine enables interrupts.

Related entries:

KM BREAK EVENT
KM DISARM BREAK

24: KM DISARM BREAK #BB48

Prevent break events from being generated.

Action:

Disarm the break mechanism. From now on the generation of break events by KM BREAK EVENT will be suppressed.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

Break events can be rearmed by calling KM ARM BREAK.

The default state of the break mechanism is disarmed, thus calling KM RESET will also disarm breaks.

This routine enables interrupts.

Related entries:

KM ARM BREAK
KM BREAK EVENT

25: KM BREAK EVENT #BB4B

Generate a break event (if armed).

Action:

Try to generate a break event.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

If the break mechanism is disarmed then no action is taken. Otherwise a break event is generated and a special marker is placed into the key buffer. This marker generates a break event token (#EF) when read from the buffer. The break mechanism is automatically disarmed after generating a break event so that multiple breaks can be avoided.

This routine may run from the interrupt path and thus does not and should not enable interrupts. Note, however, that using a LOW JUMP to call the routine (as the firmware jumpblock is set to do) does enable interrupts and so the jumpblock may not be used directly from interrupt routines.

Related entries:

KM ARM BREAK
KM DISARM BREAK

26: TXT INITIALISE #BB4E

Initialise the Text VDU.

Action:

Full initialization of the Text VDU (as used during EMS). All Text VDU variables and indirections are initialized, the previous VDU state is lost.

Entry conditions:

No conditions:

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The Text VDU indirections (TXT DRAW CURSOR, TXT UNDRAW CURSOR, TXT WRITE CHAR, TXT UNWRITE and TXT OUT ACTION) are set to their default routines.

The control code table is set up to perform the default control code actions.

The user defined character table is set to empty.

Stream 0 is selected.

All streams are set to their default states:

> The text paper (background) is set to ink 0.
> The text pen (foreground) is set to ink 1.
> The text window is set to the entire screen.
> The text cursor is enabled but turned off.
> The character write mode is set to opaque.
> The VDU is enabled.
> The graphics character write mode is turn off.
> The cursor is moved to the top left corner of the window.

The default character set and the default setting for the control code table are described in Appendices VI and VII.

Related entries:

SCR INITIALISE

27: TXT RESET #BB51

Reset the Text VDU.

Action:

Reinitialize the Text VDU indirections and the control code tables. Does not affect any other aspect of the Text VDU.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The Text VDU indirections TXT DRAW CURSOR, TXT UNDRAW CURSOR, TXT WRITE CHAR, TXT UNWRITE and TXT OUT ACTION are set to their default routines.

The control code table is set up to perform the default control code actions (see Appendix VII).

Related entries:

TXT INITIALISE

28: TXT VDU ENABLE #BB54

Allow characters to be placed on the screen.

Action:

Permit characters to be printed when requested (by calling TXT OUTPUT or TXT WR CHAR). Enabling applies to the currently selected stream. The cursor blob is also enabled (by calling TXT CUR ENABLE).

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The control code buffer used by TXT OUTPUT is emptied, any incomplete control code sequence will be lost.

Related entries:

TXT ASK STATE
TXT CUR ENABLE
TXT OUTPUT
TXT VDU DISABLE
TXT WR CHAR

29: TXT VDU DISABLE #BB57

Prevent character being placed on the screen.

Action:

Prevents characters being printed on the screen (when TXT OUTPUT or TXT WR CHAR is called). Applies to the currently selected stream. The cursor blob is also disabled (by calling TXT CUR DISABLE).

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The control code buffer used by TXT OUTPUT is emptied, any incomplete control code sequence will be lost.

In V1.1 firmware control codes are still obeyed by TXT OUTPUT. In V1.1 firmware only those control codes which are marked in the control code table will be obeyed; other control codes will be ignored (see section 4.7).

Related entries:

TXT ASK STATE
TXT CUR ENABLE
TXT OUTPUT
TXT VDU DISABLE
TXT WR CHAR

30: TXT OUTPUT #BB5A

Output a character or control code to the Text VDU.

Action:

Output characters to the screen and obey control codes (characters #00..#1F). Works on the currently selected stream.

Entry conditions:

A contains the character to send.

Exit conditions:

All registers and flags preserved.

Notes:

This routine calls the TXT OUT ACTION indirection to do the work of printing the character or obeying the control code described below.

Control codes may take up to 9 parameters. These are the characters sent following the initial control code. The characters sent are stored in the control code buffer until sufficient have been received to make up all the parameters. The control code buffer is only long enough to accept 9 parameter characters.

There is only one control code buffer for all streams. It is therefore possible to get unpredictable results if the output stream is changed midway through sending a control code sequence.

If the VDU is disabled then no characters will be printed on the screen. In V1.0 firmware all control codes will still be obeyed but in V1.1 firmware only those codes marked in the control code table as to be obeyed when the VDU is disabled will be obeyed (see section 4.7).

If the graphic character write mode is enabled then all characters and control codes are printed using the Graphics VDU routine, GRA WR CHAR, and are not obeyed.

Characters are written in the same way that TXT WR CHAR writes characters.

Related entries:

GRA WR CHAR
TXT OUT ACTION
TXT SET GRAPHIC
TXT VDU DISABLE
TXT VDU ENABLE
TXT WR CHAR

31: TXT WR CHAR #BB5D

Write a character to the screen.

Action:

Print a character on the screen at the cursor position of the currently selected stream. Control codes (characters #00..#1F) are printed and not obeyed.

Entry conditions:

A contains the character to print.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

If the VDU is disabled then no character will be printed.

Before printing the character the cursor position is forced to lie within the text window (see TXT VALIDATE). After printing the character the cursor is moved right one character.

To put the character on the screen this routine calls the TXT WRITE CHAR indirection.

Related entries:

GRA WR CHAR
TXT OUTPUT
TXT RD CHAR
TXT WRITE CHAR

32: TXT RD CHAR #BB60

Read a character from the screen.

Action:

Read a character from the screen at the cursor position of the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

If a recognisable character was found:

Carry true.
A contains the character read.

If no recognisable character was found:

Carry false.
A contains zero.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

In V1.1 firmware the cursor position is forced legal (inside the window) before the character is read. This may cause the screen to roll. The same is not true in V1.0 firmware where the cursor position is not forced legal and steps must be taken to avoid reading characters from outside the window.

The read is performed by comparing the matrix found on the screen with the matrices used to generate characters. As a result changing a character matrix, changing the pen or paper inks, or changing the screen (e.g. drawing a line through a character) may make the character unreadable.

To actually read the character from the screen the TXT UNWRITE indirection is called.

Special precautions are taken against generating inverse space (character #8F). Initially the character is read assuming that the background to the character was written in the current paper ink and treating any other ink as foreground. If this fails to generate a recognisable character or it generates inverse space then another try is made by assuming that the foreground to the character was written in the current pen ink and treating any other ink as background.

The characters are scanned starting with #00 and finishing with #FF.

Related entries:

TXT UNWRITE
TXT WR CHAR

33: TXT SET GRAPHIC #BB63

Turn on or off the Graphics VDU write character option.

Action:

Enable or disable graphic character writing on the currently selected stream.

Entry conditions:

If graphic writing is to be turned on:
  A must be non-zero.

If the graphic writing is to be turned off:
  A must contain zero.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

When graphic character writing is enabled then all characters sent to TXT OUTPUT are printed using the Graphics VDU (see GRA WR CHAR) rather then the Text VDU (see TXT WR CHAR). Also all control codes are printed rather than obeyed. Characters sent to TXT WR CHAR will be printed as normal.

Character printing is not prevented by disabling the Text VDU (with TXT VDU DISABLE) if graphic character writing is enabled.

Related entries:

GRA WR CHAR
TXT OUTPUT

34: TXT WIN ENABLE #BB66

Set the size of the current text window.

Action:

Set the boundaries of the window on the currently selected stream. The edges are the first and last character columns inside the window and the first and last character rows inside the window.

Entry conditions:

H contains the physical column of one edge.

D contains the physical column of the other edge.

L contains the physical row of one edge.

E contains the physical row of the other edge.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The edge positions are given in physical screen coordinates. i.e. Row 0, column 0 is the top left corner of the screen and the coordinates are signed numbers.

The window is truncated, if necessary, so that it fits on the screen.

The left column of the window is taken to be the smaller of H and D. The top row of the window is taken to be the smaller of L and E.

The cursor is moved to the top left corner of the window.

The window is not cleared.

If the window covers the whole screen then when the window is rolled the hardware roll routine (see SCR HW ROLL) will be used. If the window covers less than the whole screen the software roll routine (see SCR SW ROLL) will be used.

The default text window covers the whole screen and is set up when TXT INITIALISE or SCR SET MODE is called.

Related entries:

TXT GET WINDOW
TXT VALIDATE

35: TXT GET WINDOW #BB69

Get the size of the current window.

Action:

Get the boundaries of the window on the currently selected stream and whether it covers the whole screen.

Entry conditions:

No conditions.

Exit conditions:

If the window covers the whole screen:

Carry false.

If the window covers less than the whole screen:

Carry true.

Always:

H contains the leftmost column in the window.
D contains the rightmost column in the window.
L contains the topmost row in the window.
E contains the bottommost row in the window.
A corrupt.
All other registers preserved.

Notes:

The boundaries of the window are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen.

The boundaries returned by this routine may not be the same as those set when TXT WIN ENABLE was called because the window is truncated to fit the screen.

Related entries:

TXT VALIDATE
TXT WIN ENABLE

36: TXT CLEAR WINDOW #BB6C

Clear current window.

Action:

Clear the text window of the currently selected stream to the paper ink of the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The cursor is moved to the top left corner of the window.

Related entries:

GRA CLEAR WINDOW
SCR CLEAR
TXT SET PAPER
TXT WIN ENABLE

37: TXT SET COLUMN #BB6F

Set cursor horizontal position.

Action:

Move the current position of the currently selected stream to a new column. The cursor blob will be removed from the current position and redrawn at the new position (if the cursor is enabled and turned on).

Entry conditions:

A contains the required logical column for the cursor.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The required column is given in logical coordinates. i.e. Column 1 is the leftmost column of the window.

The cursor may be moved outside the window. However, it will be forced to lie inside the window before any character is written by the Text VDU (see TXT VALIDATE) or the cursor blob is drawn.

Related entries:

TXT GET CURSOR
TXT SET CURSOR
TXT SET ROW

38: TXT SET ROW #BB72

Set cursor vertical position.

Action:

Move the current position of the currently selected stream to a new row. The cursor blob will be removed from the current position and redrawn at the new position (if the cursor is enabled and turned on).

Entry conditions:

A contains the required logical row for the cursor.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The required row is given in logical coordinates. i.e. Row 1 is the topmost row of the window.

The cursor may be moved outside the window. However, it will be forced to lie inside the window before any characters is written by the Text VDU (see TXT VALIDATE) or the cursor blob is drawn.

Related entries:

TXT GET CURSOR
TXT SET COLUMN
TXT SET CURSOR

39: TXT SET CURSOR #BB75

Set cursor position.

Action:

Move the current position of the currently selected stream to a new row and column. The cursor blob will be removed from the current position and redrawn at the new position (if the cursor is enabled and turned on).

Entry conditions:

H contains the required logical column.

L contains the required logical row.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The required row is given in logical coordinates. i.e. Row 1, column 1 is the top left corner of the window.

The cursor may be moved outside the window. However, it will be forced to lie inside the window before any characters is written by the Text VDU (see TXT VALIDATE) or the cursor blob is drawn.

Related entries:

TXT GET CURSOR
TXT SET COLUMN
TXT SET ROW

40: TXT GET CURSOR #BB78

Ask current cursor position.

Action:

Get the current location of the cursor and a count of the number of times the window of the currently selected stream has rolled.

Entry conditions:

No conditions.

Exit conditions:

H contains the logical cursor column.

L contains the logical cursor row.

A contains the current roll count.

Flags corrupt.

All other registers preserved.

Notes:

The cursor position is given in logical coordinates. i.e. Row 1, column 1 is the top left corner of the window.

The roll count passed out has no absolute meaning. It is decremented when the window is rolled up and is incremented when the window is rolled down. It may be used to determine whether the window has rolled by comparing it with a previous value.

The position reported may not be inside the window and is, therefore, not necessarily the position at which the next character will be printed. Use TXT VALIDATE to check this.

Related entries:

TXT SET COLUMN
TXT SET CURSOR
TXT SET ROW
TXT VALIDATE

41: TXT CUR ENABLE #BB7B

Allow cursor display - user.

Action:

Allow the cursor blob for the currently selected stream to be placed on the screen. The cursor blob will be placed on the screen immediately unless the cursor is turned off (see TXT CUR OFF).

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

Cursor enabling and disabling is intended for use by the user. It is also used when the VDU is disabled (TXT VDU ENABLE and TXT VDU DISABLE).

Related entries:

TXT ASK STATE
TXT CUR DISABLE
TXT CUR ON
TXT DRAW CURSOR
TXT UNDRAW CURSOR

42: TXT CUR DISABLE #BB7E

Disallow cursor display - user.

Action:

Prevent the cursor blob for the currently selected stream from being placed on the screen. The cursor blob will be removed from the screen immediately if it is currently there.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

Cursor enabling and disabling is intended for use by the user. It is also used when the VDU is disabled (TXT VDU ENABLE and TXT VDU DISABLE).

Related entries:

TXT ASK STATE
TXT CUR ENABLE
TXT CUR OFF
TXT DRAW CURSOR
TXT UNDRAW CURSOR

43: TXT CUR ON #BB81

Allow cursor display - system.

Action:

Allow the cursor blob for the currently selected stream to be placed on the screen. The cursor blob will be placed on the screen immediately unless the cursor in disabled (see TXT CUR DISABLE).

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

Turning the cursor on and off is intended for use by system ROMs.

Related entries:

TXT ASK STATE
TXT CUR ENABLE
TXT CUR OFF
TXT DRAW CURSOR
TXT UNDRAW CURSOR

44: TXT CUR OFF #BB84

Disallow cursor display - system.

Action:

Prevent the cursor blob for the currently selected stream from being placed on the screen. The cursor blob will be removed from the screen immediately if it is currently there.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

Turning the cursor on and off is intended for use by system ROMs.

Related entries:

TXT ASK STATE
TXT CUR DISABLE
TXT CUR ON
TXT DRAW CURSOR
TXT UNDRAW CURSOR

45: TXT VALIDATE #BB87

Check if a cursor position is within the window.

Action:

Check a screen position to see if it lies within the current window. If it does not then determine the position where a character would be printed after applying the rules for forcing the screen position inside the window.

Entry conditions:

H contains the logical column of the position to check.

L contains the logical row of the position to check.

Exit conditions:

If printing at the position would not cause the window to roll:

Carry true.
B corrupt.

If printing at the position would cause the window to roll up:

Carry false.
B contains #FF.

If printing at the position would cause the window to roll down:

Carry false.
B contains #00.

Always:

H contains the logical column at which a character would be printed.
L contains the logical row at which a character would be printed.
A and other flags corrupt.
All other registers preserved.

Notes:

The position on the screen are given in logical coordinates. i.e. Row 1, column 1 is the top left corner of the window.

Before writing a character or putting the cursor blob on the screen the Text VDU validates the current position, performs any required roll then writes at the appropriate position.

The algorithm to work out the position to print at, from the position to check, is as follows:

1/ If the position is right of the right edge of the window it is moved to the left edge of the window on the next line.

2/ If the position is left of the left edge of the window it is moved to the right edge of the window on the previous line.

3/ If the position is now above the top edge of the window then it is moved to the top edge of the window and the window need rolling downwards.

4/ If the position is now below the bottom edge of the window it is moved to the bottom edge of the window and the window needs rolling upwards.

Related entries:

SCR HW ROLL
SCR SW ROLL
TXT GET CURSOR

46: TXT PLACE CURSOR #BB8A

Put a cursor blob on the screen.

Action:

Put a cursor blob on the screen at the cursor position for the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

TXT PLACE CURSOR is provided to allow the user to run multiple cursors in a window. The indirection TXT DRAW CURSOR should be called for merely placing the normal cursor blob on the screen. Higher level routines, such as TXT OUTPUT and TXT SET CURSOR, automatically remove and place the normal cursor when appropriate, the user must deal with any other cursors.

It is not safe to call TXT PLACE CURSOR twice at a particular screen position without calling TXT REMOVE CURSOR in between because this may leave a spurious cursor blob on the screen when the cursor position is moved.

The cursor position is forced to be inside the window before the cursor blob is drawn.

The cursor blob is an inverse patch formed by exclusive-oring the contents of the screen at the cursor position with the exclusive-or of the current pen and paper inks.

Related entries:

TXT DRAW CURSOR
TXT REMOVE CURSOR

47: TXT REMOVE CURSOR #BB8D

Take a cursor blob off the screen.

Action:

Take the cursor blob off the screen at the cursor position of the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

TXT REMOVE CURSOR is provided to allow the user to run multiple cursors in a window. The indirection TXT UNDRAW CURSOR should be called for merely removing the normal from the screen. Higher level routines, such as TXT OUTPUT and TXT SET CURSOR, automatically remove and place the normal cursor when appropriate, the user must deal with any other cursors.

TXT REMOVE CURSOR should only be used to remove a cursor placed on the screen by calling TXT PLACE CURSOR. The cursor should be removed when the cursor position is to be changed (rolling the window implicitly changes the cursor position) or the screen is to read or written. Incorrect use of this routine may result in a spurious cursor blob being generated.

The cursor position is forced to be inside the window before the cursor blob is removed (this should not matter as TXT PLACE CURSOR has already done this).

The cursor blob is an inverse patch formed by exclusive-oring the contents of the screen at the cursor position with the exclusive-or of the current pen and paper inks.

Related entries:

TXT PLACE CURSOR
TXT UNDRAW CURSOR

48: TXT SET PEN #BB90

Set ink for writing characters.

Action:

Set the pen ink for the currently selected stream. This is the ink that is used for writing characters (the foreground ink).

Entry conditions:

A contains ink to use.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The ink is masked to bring it within the range of legal inks for the current screen mode. That is with #0F in mode 0, #03 in mode 1 and #01 in mode 2.

The cursor blob will be redrawn using the new ink (if enabled).

Related entries:

GRA SET PEN
SCR SET INK
TXT GET PEN
TXT SET PAPER

49: TXT GET PEN #BB93

Get ink for writing characters.

Action:

Ask what the pen ink is set to for the currently selected stream. This is the ink used for writing characters (foreground ink).

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

GRA GET PEN
SCR GET INK
TXT GET PAPER
TXT SET PEN

50: TXT SET PAPER #BB96

Set ink for writing text background.

Action:

Set the text paper ink for the currently selected stream. This is the ink used for writing the background to characters and for clearing the text window.

Entry conditions:

A contains the ink to use.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The ink is masked to bring it within the range of legal inks for the current screen mode. That is with #0F in mode 0, #03 in mode 1 and #01 in mode 2.

The cursor blob will be redrawn using the new ink (if enabled).

This ink will be used when clearing areas of the text window (by TXT CLEAR WINDOW and certain control codes).

This routine does not clear the text window.

Related entries:

GRA GET PAPER
SCR SET INK
TXT GET PAPER
TXT SET PEN

51: TXT GET PAPER #BB99

Get ink for writing background.

Action:

Ask what the paper ink is set to for the currently selected stream. The ink used for writing the background to characters and for clearing the text window.

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

GRA GET PAPER
SCR GET INK
TXT GET PEN
TXT SET PAPER

52: TXT INVERSE #BB9C

Swap current pen and paper inks over.

Action:

Exchange the text pen and paper (foreground and background) inks for the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

In V1.1 firmware the cursor blob is removed and replaced and so the current position is forced legal (inside the window) which may cause the window to roll. In V1.0 firmware the cursor blob is not redrawn and so it should be on the screen when this routine is called.

Related entries:

TXT SET PAPER
TXT SET PEN

53: TXT SET BACK #BB9F

Allow or disallow background being written.

Action:

Set character write mode to opaque or transparent for the currently selected stream. Opaque mode writes background with the character. Transparent mode writes the character on top of the current contents of the screen.

Entry conditions:

If the background is to be written (opaque mode):

A must be zero.

If background is not to be written (transparent mode):

A must be non-zero.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

Writing in transparent mode is intended for annotating diagrams and similar applications. It can have unfortunate effects if it is used generally because overwriting a character will not remove the character underneath thus creating an incomprehensible jumble on the screen.

Setting the character write mode does not affect the Graphics VDU. In V1.1 firmware the routine GRA SET BACK sets the equivalent graphics background write mode.

Related entries:

GRA SET BACK
TXT GET BACK
TXT WR CHAR
TXT WRITE CHAR

54: TXT GET BACK #BBA2

Ask if background is being written.

Action:

Get the character write mode for the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

If background is to be written (opaque mode):
    A contains zero.
If background is not to be written (transparent mode):
    A contains non-zero.
Always:
    DE, HL and flags corrupt.
    All registers preserved.

Notes:

This only applies to the Text VDU, the Graphics VDU always writes opaque.

Related entries:

TXT SET BACK

55: TXT GET MATRIX #BBA5

Get the address of a character matrix.

Action:

Calculate a pointer to the matrix for a character and determine if it is a user defined matrix.

Entry conditions:

A contains the character whose matrix is to be found.

Exit conditions:

If the matrix in the user defined matrix table:

Carry true.

If the matrix is in the lower ROM:

Carry false.

Always:

HL contains the address of the matrix.
A and other flags corrupt.
All other registers preserved.

Notes:

The matrix may be in RAM or in ROM. The Text VDU assumes that the appropriate ROMs are enabled or disabled when it calls this routine to get the matrix for a character. (The lower ROM is on, the upper ROM is normally off).

The matrix is stored as an 8 byte bit significant vector. The first byte describes the top line of the character and the last byte the bottom line. Bit 7 of a byte refers to the leftmost pixel of a line and bit 0 to the rightmost pixel. If a bit is set in the matrix then the pixel should be written in the pen ink. If the bit is not set then the pixel should either be written in the paper ink or left alone (depending on the character write mode).

Related entries:

TXT SET MATRIX

56: TXT SET MATRIX #BBA8

Set a character matrix.

Action:

Set the matrix for a user defined character. If the character is not user defined then no action is taken.

Entry conditions:

A contains the character whose matrix is to be set.

HL contains the address of the matrix to set.

Exit conditions:

If the character is user definable:

Carry true.

If the character is not user definable:

Carry false.

Always:

A, BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

The matrix is stored as an 8 byte bit significant vector. The first byte describes the top line of the character and the last byte the bottom line. Bit 7 of a byte refers to the leftmost pixel of a line and bit 0 to the rightmost pixel. If a bit is set in the matrix then the pixel should be written in the pen ink. If a bit is not set then the pixel should either be written in the paper ink or left alone (depending whether the character write mode is opaque or transparent currently).

The matrix is copied from the area given into the character matrix table without using RAM LAMs thus the matrices can be set from ROM providing it is enabled. (Note however that the jumpblock disables the upper ROM).

Altering a character matrix changes the matrix for all streams. It does not alter any character on the screen; it changes what will be placed on the screen the next time the character is written.

Related entries:

TXT GET MATRIX
TXT SET M TABLE

57: TXT SET M TABLE #BBAB

Set the user defined matrix table address.

Action:

Set the user defined matrix table and the number of characters in the table. The table is initialized with the current matrix settings.

Entry conditions:

DE contains the first character in the table.

HL contains the address of the start of the new table.

Exit conditions:

If there was no user defined matrix table before:

Carry false.
A and HL corrupt.

If there was a user defined matrix table before:

Carry true.
A contains the first character in the old table.
HL contains the address of the old table.

Always:

BC, DE and other flags corrupt.
All other registers preserved.

Notes:

If the first character specified is in the range 0..255 then the matrices for all characters between that character and character 255 are to be stored in the user defined table.

If the first character specified is not in the range 0..255 then the user defined matrix table is deemed to contain no matrices (and the table address passed is ignored).

The table must be (256 - first char) * 8 bytes long. The matrices are stored in the table in ascending order. The table is initialized with the current matrix settings, whether they were previously in RAM or in the ROM.

The table should not be located in RAM underneath a ROM.

It is permissible for the new and old matrix tables to overlap (thus allowing the table to be extended or contracted) providing that matrices in the new table occupy an address earlier to the address that they occupied in the old table.

All streams share the matrix table so any changes to it will be reflected on all streams.

Related entries:

TXT GET M TABLE
TXT SET MATRIX

58: TXT GET M TABLE #BBAE

Get user defined matrix table address.

Action:

Get the address of the current user defined matrix table and the first character in the table.

Entry conditions:

No conditions.

Exit conditions:

If there is no user defined matrix table:

    Carry false.
    A and HL corrupt.

If there is a user defined matrix table:

    Carry true.
    A contains the first character in the table.
    HL contains the address of the start of the table.

Always:

    Other flags corrupt.
    All other registers preserved.

Notes:

The matrices for characters between the first character and 255 are stored in the table in ascending order. Each matrix is 8 bytes long.

Related entries:

TXT GET MATRIX
TXT SET M TABLE

59: TXT GET CONTROLS #BBB1

Fetch address of control code table.

Action:

Get the address of the control code table.

Entry conditions:

No conditions.

Exit conditions:

HL contains the address of the control code table.

All other registers and flags preserved.

Notes:

All streams share one control code table so that any changes made to the table will affect all streams.

The control code table has a 3 byte entry for each control code. The entries are stored in ascending order, so the entry for #00 is first and that for #1F is last. The first byte of each entry is the number of parameters the control code requires, the other two bytes are the address of the routine to call to process the control code when all its parameters have been received. The routine must be located in the central 32K of RAM and it must obey the following interface:

Entry:

A contains the last character added to the buffer.
B contains the length of the buffer (including the control code).
C contains the same as A.
HL contains the address of the control code buffer (points at the control code).

Exit:

AF, BC, DE, HL corrupt.
All other registers preserved.

As the control buffer only has space to store 9 parameter characters the number of parameters required should be limited to 9 or fewer.

The control code table is reinitialized to its default routines when TXT RESET is called.

In V1.1 firmware the first byte of each entry also specifies whether the control codes is to be disabled when the VDU is disabled or whether it is always to be obeyed. Bit 7 of the byte is set if the code is to be disabled.

Related entries:

TXT OUTPUT

60: TXT STR SELECT #BBB4

Select a Text VDU stream.

Action:

Make a given stream the currently selected stream (if it isn't already).

Entry conditions:

A contains the required stream.

Exit conditions:

A contains the previously selected stream.

HL and flags corrupt.
All other registers preserved.

Notes:

The requested stream number is masked (with #07) to make it into a legal stream number.

Many attributes of the Text VDU may be set independently on different streams. It is important to ensure that the correct stream is selected when any of these are altered. These attributes are:

Pen ink.
Paper ink.
Cursor position.
Window limits.
Cursor enable/disable.
Cursor on/off. VDU enable/disable.
Character write mode.
Graphics character write mode.

If the stream is already selected then this routine returns quickly. It is not unreasonable to repeatedly select a stream (before each character sent, for example).

Related entries:

TXT OUTPUT

61: TXT SWAP STREAMS #BBB7

Swap the states of two streams.

Action:

The stream descriptors for two streams are exchanged. The currently selected stream number remains the same (although its descriptor may have been altered).

Entry conditions:

B contains a stream number.

C contains another stream number.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The stream numbers passed are masked (with #07) to that they are legal stream numbers.

The attributes that are exchanged are:

Pen ink.
Paper ink.
Cursor position.
Window limits.
Cursor enable/disable.
Cursor on/off.
VDU enable/disable.
Character write mode.
Graphics character write mode.

Related entries:

TXT STR SELECT

62: GRA INITIALISE #BBBA

Initialize the Graphics VDU.

Action:

The Graphics VDU is fully initialized (as during EMS). All Graphic VDU variables and indirections are set to their default values.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The full operation is:

> Set the Graphics VDU indirections (GRA PLOT, GRA TEST and GRA LINE) to their default routines.
> Set the graphics paper to ink 0.
> Set the graphics pen to ink 1.
> Set the user origin to the bottom left corner of the screen.
> Move the current position to the user origin.
> Set the graphics window to cover the whole screen.
> The graphics background write mode is set to opaque.
> The line mask is set to #FF and the first pixel of lines are plotted.
> The graphics window is not cleared.

Related entries:

GRA DEFAULT
GRA RESET
SCR INITIALISE

63: GRA RESET #BBBD

Reset the Graphics VDU.

Action:

Re-initialize the Graphics VDU indirections to their default routines and set default modes.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

Sets the Graphics VDU indirections (GRA PLOT, GRA TEST and GRA LINE) to their default routines. V1.1 firmware also sets the graphics background mode to opaque, sets the line mask to #FF and sets the first pixel of lines to be plotted.

Related entries:

GRA DEFAULT
GRA INITIALISE

64: GRA MOVE ABSOLUTE #BBC0

Move to an absolute position.

Action:

Move the current position to an absolute position.

Entry conditions:

DE contains the required user X coordinate.
HL contains the required user Y coordinate.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The new position is given in user coordinates. i.e. Relative to the user origin.

The new position can be outside the graphics window.

The Graphics VDU plotting, testing and line drawing routines all move the current graphics position to the point (or endpoint) specified automatically.

Related entries:

GRA ASK CURSOR
GRA MOVE RELATIVE

65: GRA MOVE RELATIVE #BBC3

Move relative to current position.

Action:

Move the current position to relative to its current position.

Entry conditions:

DE contains a signed X offset.
HL contains a signed Y offset.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The new position can be outside the graphics window.

The Graphics VDU plotting, testing and line drawing routines all move the current graphics position to the point (or endpoint) specified automatically.

Related entries:

GRA ASK CURSOR
GRA MOVE ABSOLUTE

66: GRA ASK CURSOR #BBC6

Get the current position.

Action:

Ask where the current graphics position is.

Entry conditions:

No conditions.

Exit conditions:

DE contains the user X coordinate.
HL contains the user Y coordinate.

AF corrupt.
All other registers preserved.

Notes:

The new position is given in user coordinates. i.e. Relative to the user origin.

The Graphics VDU plotting, testing and line drawing routines all move the current graphics position to the point (or endpoint) specified automatically. Thus, the position returned is probably where the last point was plotted or tested.

Related entries:

GRA MOVE ABSOLUTE
GRA MOVE RELATIVE

67: GRA SET ORIGIN #BBC9

Set the origin of the user coordinates.

Action:

Set the location of the user origin and move the current position there.

Entry conditions:

DE contains the standard X coordinate of the origin.
HL contains the standard Y coordinate of the origin.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The origin position is given is standard coordinates in which (0,0) is the bottom left corner of the screen.

The default origin position is at (0,0). Whenever the screen mode is changed, by calling SCR SET MODE, the origin is restored to its default position.

Related entries:

GRA FROM USER
GRA GET ORIGIN

68: GRA GET ORIGIN #BBCC

Get the origin of the user coordinates.

Action:

Ask where the user coordinate origin is located.

Entry conditions:

No conditions.

Exit conditions:

DE contains the standard X coordinate of the origin.
HL contains the standard Y coordinate of the origin.

All other registers preserved.

Notes:

The origin position is given is standard coordinates in which (0,0) is the bottom left corner of the screen.

Related entries:

GRA SET ORIGIN

69: GRA WIN WIDTH #BBCF

Set the right and left edges of the graphics window.

Action:

Set the horizontal position of the graphics window. The left and right edges are respectively the first and last points that lie inside the window horizontally.

Entry conditions:

DE contains the standard X coordinate of one edge.

HL contains the standard X coordinate of the other edge.

Exit conditions:

AF, BC, DE and HL corrupt.

All registers preserved.

Notes:

The window edges are given in standard coordinates in which (0,0) is the bottom left corner of the screen and coordinates are signed 16 bit numbers.

The left edge of the window is deemed to be the smaller of the two edges supplied.

The window will be truncated, if necessary, to make it fit the screen. The edges are moved to screen byte boundaries so that the window only contains whole bytes (the left edge is moved left, the right edge is moved right). This moves the coordinates of the edges as follows in the various modes:

Mode	Left Edge	Right Edge
0	Multiple of 2	Multiple of 2 minus 1
1	Multiple of 4	Multiple of 4 minus 1
2	Multiple of 8	Multiple of 8 minus 1

The default window covers the whole screen. Whenever the screen mode is changed the window is restored to its default size.

All Graphics VDU point plotting and line drawing routines test whether the points they are about to plot lie inside the window; if they are not then the points are not plotted.

Related entries:

GRA GET W WIDTH
GRA WIN HEIGHT

70: GRA WIN HEIGHT #BBD2

Set the top and bottom edges of the graphics window.

Action:

Set the vertical position of the graphics window. The top and bottom edges are respectively the last and first points that lie inside the window vertically.

Entry conditions:

DE contains the standard Y coordinate of one edge.

HL contains the standard Y coordinate of the other edge.

Exit conditions:

AF, BC, DE and HL corrupt.

All registers preserved.

Notes:

The window edges are given in standard coordinates in which (0,0) is the bottom left corner of the screen and coordinates are signed 16 bit numbers.

The top edge will be deemed to be the higher of the two edges supplied.

The window will be truncated, if necessary, to make it fit the screen. The edges will be moved to lie on screen line boundaries so that only whole screen lines are included in the window (the top edge will be moved up, the bottom edge will be moved down). This moves the bottom edge to an even coordinate and the top edge to an odd coordinate.

The default window covers the whole screen. Whenever the screen mode is changed the window is restored to its default size.

All Graphics VDU point plotting and line drawing routines test whether the points they are about to plot lie inside the window; if they are not then the points are not plotted.

Related entries:

GRA GET W HEIGHT
GRA WIN WIDTH

71: GRA GET W WIDTH #BBD5

Get the left and right edges of the graphics window.

Action:

Ask the horizontal position of the graphics window. The left and right edges are respectively the first and last points that lie inside the window horizontally.

Entry conditions:

No conditions.

Exit conditions:

DE contains the standard X coordinate of the left edge of the window.
HL contains the standard X coordinate of the right edge of the window.

AF corrupt.
All other registers preserved.

Notes:

The window edges are given in standard coordinates in which (0,0) is the bottom left corner of the screen.

The edges may not be exactly the same as those that were set using GRA WIN WIDTH as the window is truncated to fit the screen, and the edges are moved to screen byte boundaries so that the window only contains whole bytes.

Related entries:

GRA GET W HEIGHT
GRA IN WIDTH

72: GRA GET W HEIGHT #BBD8

Get the top and bottom edges of the graphics window.

Action:

Ask the vertical position of the graphics window. The top and bottom edges are respectively the last and first points that lie inside the graphics window vertically.

Entry conditions:

No conditions.

Exit conditions:

DE contains the standard Y coordinate of the top edge of the window.
HL contains the standard Y coordinate of the bottom edge of the window.

AF corrupt.
All other registers preserved.

Notes:

The window edges are given in standard coordinates. i.e. With (0,0) being the bottom left corner of the screen.

The edges may not be exactly the same as that passed to GRA WIN HEIGHT as the window is truncated to fit the screen, and the edges are moved to lie on screen line boundaries so that only whole screen lines are included in the window.

Related entries:

GRA GET W WIDTH
GRA WIN HEIGHT

73: GRA CLEAR WINDOW #BBDB

Clear the graphics window.

Action:

Clear the graphics window to the graphics paper ink.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The current graphics position is moved to the origin of the user coordinates.

Related entries:

GRA SET PAPER
GRA WIN HEIGHT
GRA WIN WIDTH
SCR CLEAR
TXT CLEAR WINDOW

74: GRA SET PEN #BBDE

Set the graphics plotting ink.

Action:

Set the graphics pen ink. This is the ink by the Graphics VDU for plotting points, drawing lines and writing characters.

Entry conditions:

A contains the required ink.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The ink is masked to bring it in the range of inks for the current screen mode. In mode 0 the mask is #0F, in mode 1 it is #03 and in mode 2 it is #01.

In V1.1 firmware the graphics pen ink is taken to delimit the edge of the area to fill when flood filling areas of the screen.

Related entries:

GRA GET PEN
GRA SET PAPER
SCR SET INK
TXT SET PEN

75: GRA GET PEN #BBE1

Get the current graphics plotting ink.

Action:

Ask what the current graphics pen ink is set to. This is the ink used by the Graphics VDU for plotting points, drawing lines and writing characters.

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

GRA GET PAPER
GRA SET PEN
SCR GET INK
TXT GET PEN

76: GRA SET PAPER #BBE4

Set the graphics background ink.

Action:

Set the graphics paper ink.

Entry conditions:

A contains the required ink.

Exit conditions:

AF corrupt.

All registers preserved.

Notes:

The ink is masked to bring it in the range of inks for the current screen mode. In mode 0 the mask is #0F, in mode 1 it is #03 and in mode 2 it is #01.

The paper ink is the ink that is used for clearing the graphics window, and writing the background to characters. It is assumed to cover everywhere outside the graphics window when testing points.

In V1.1 firmware the graphics paper ink is used to plot pixels corresponding to a zero bit in the line mask when drawing lines.

Related entries:

GRA GET PAPER
GRA SET PEN
SCR GET INK
TXT SET PAPER

77: GRA GET PAPER #BBE7

Get the current graphics background ink.

Action:

Ask what the current graphics paper ink is set to.

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

The paper ink is the ink that is used for clearing the graphics window, and writing background to characters. It is assumed to cover everywhere outside the graphics window when testing points.

Related entries:

GRA GET PEN
GRA SET PAPER
SCR GET INK
TXT GET PAPER

78: GRA PLOT ABSOLUTE #BBEA

Plot a point at an absolute position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the point is plotted in the current graphics pen ink using the current graphics write mode. If the point lies outside the graphics window then no action is taken.

Entry conditions:

DE contains the user X coordinate to plot at.

HL contains the user Y coordinate to plot at.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The position to plot at is given in user coordinates. i.e. Relative to the user origin.

This routine calls the GRA PLOT indirection to plot the point. In its turn GRA PLOT calls the SCR WRITE indirection to set the pixel (if it is in the window).

Related entries:

GRA PLOT
GRA PLOT RELATIVE
GRA TEST ABSOLUTE

79: GRA PLOT RELATIVE #BBED

Plot a point relative to the current position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the point is plotted in the current graphics pen ink using the current graphics write mode. If the point lies outside the graphics window then no action is taken.

Entry conditions:

DE contains a signed X offset.

HL contains a signed Y offset.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The position to plot at is given in relative coordinates. i.e. Relative to the current graphics position.

This routine calls the GRA PLOT indirection to plot the point. In its turn GRA PLOT calls the SCR WRITE indirection to set the pixel (if it is in the window).

Related entries:

GRA PLOT
GRA PLOT ABSOLUTE
GRA TEST RELATIVE

80: GRA TEST ABSOLUTE #BBF0

Test a point at an absolute position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the pixel is read from the screen and the ink it is set to is decoded and returned. If the position lies outside the graphics window then the current paper ink is returned.

Entry conditions:

DE contains the user X coordinate to test at.
HL contains the user Y coordinate to test at.

Exit conditions:

A contains the ink of the specified point (or the graphics paper ink).

BC, DE, HL and flags corrupt.
All other registers preserved.

Notes:

The position to test is given in user coordinates. i.e. Relative to the user origin.

This routine calls the GRA TEST indirection to test the point. In its turn GRA TEST calls the SCR READ indirection to test the pixel (if it is in the window).

Related entries:

GRA PLOT ABSOLUTE
GRA TEST
GRA TEST RELATIVE

81: GRA TEST RELATIVE #BBF3

Test a point relative to the current position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the pixel is read from the screen and the ink it is set to is decoded and returned. If the position is outside the graphics window then the current paper ink is returned.

Entry conditions:

DE contains a signed X offset.

HL contains a signed Y offset.

Exit conditions:

A contains the ink of the specified point (or the graphics paper ink).

BC, DE, HL and flags corrupt.

All other registers preserved.

Notes:

The position to test is given in relative coordinates. i.e. Relative to the current graphics position.

This routine calls the GRA TEST indirection to test the point. In its turn GRA TEST calls the SCR READ indirection to test the pixel (if it is in the window).

Related entries:

GRA PLOT RELATIVE
GRA TEST
GRA TEST ABSOLUTE

82: GRA LINE ABSOLUTE #BBF6

Draw a line to an absolute position.

Action:

Move the current graphics position to the endpoint supplied. All points between this position and the previous graphics position that lie inside the graphics window may be plotted. Points that lie outside the graphics window are ignored.

Entry conditions:

DE contains the user X coordinate of the endpoint.

HL contains the user Y coordinate of the endpoint.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The position of the end of the line is given in user coordinates. i.e. Relative to the user origin.

In V1.0 firmware the points will be plotted in the current graphics pen ink using the current graphics write mode.

In V1.1 firmware the setting of the line mask determines how pixels on the line will be plotted. The line mask is bit significant and is used repeatedly in the order bit 7, bit 6 down to bit 0 for each pixel in the line. If the bit is one then the pixel is plotted in the graphics pen ink using the current graphics write mode. If the bit is zero then the action taken depends on the graphics background write mode. If the background mode is opaque then the pixel is plotted in the graphics paper ink using the current graphics write mode. If the background mode is transparent then the pixel is not plotted.

In V1.1 firmware the first pixel of the line (that at the previous graphics position) is not plotted if the first point plotting mode is set false.

This routine calls the GRA LINE indirection to draw the line. In its turn GRA LINE calls the SCR WRITE indirection to write the pixels (for pixels in the graphics window).

Related entries:

GRA LINE
GRA LINE RELATIVE
GRA SET BACK
GRA SET FIRST
GRA SET LINE MASK

83: GRA LINE RELATIVE #BBF9

Draw a line relative to the current position.

Action:

Move the current graphics position to the endpoint supplied. All points between this position and the previous graphics position that lie inside the graphics window may be plotted. Points that lie outside the graphics window are ignored.

Entry conditions:

DE contains the signed X offset of the endpoint.

HL contains the signed Y offset of the endpoint.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The position of the end of the line is given in relative coordinates. i.e. Relative to the current graphics position.

In V1.0 firmware the points will be plotted in the current graphics pen ink using the current graphics write mode.

In V1.1 firmware the setting of the line mask determines how the pixels on the line will be plotted. The line mask is bit significant and is used repeatedly in the order bit 7, bit 6 down to bit 0 for each pixel in the line. If the bit is one then the pixel is plotted in the graphics pen ink using the current graphics write mode. If the bit is zero then the action taken depends on the graphics background write mode. If the background mode is opaque then the pixel is plotted in the graphics paper ink using the current graphics write mode. If the background mode is transparent then the pixel is not plotted.

In V1.1 firmware the first pixel of the line (that at the previous graphics position) is not plotted if the first point plotting mode is set false.

This routine calls the GRA LINE indirection to draw the line. In its turn GRA LINE calls the SCR WRITE indirection to write the pixels (for pixels in the graphics window).

Related entries:

GRA LINE
GRA LINE ABSOLUTE
GRA SET BACK
GRA SET FIRST
GRA SET LINE MASK

84: GRA WR CHAR #BBFC

Put a character on the screen at the current graphics position.

Action:

Write a character on the screen at the current graphics position.

Entry conditions:

A contains the character to write.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The character is written with its top left corner being the current graphics position.

All characters are printed, even control codes (characters #00..#1F).

The current position is moved right by the width of the character (ready for another character to be written). In mode 0 this move is 32 points right, in mode 1 the move is 16 points right and in mode 2 it is 8 points.

The character will be plotted in the graphics pen ink. In the case of V1.0 firmware, or V1.1 firmware when the background write mode is set to opaque, the background to the character will be plotted in the graphics paper ink. In the case of V1.1 firmware when the background write mode is set transparent the background pixels are not plotted. Pixels in the character that lie outside the graphics window will not be plotted. The pixels are plotted using the SCR WRITE indirection so they are written using the current graphics write mode.

Related entries:

GRA SET BACK
TXT SET GRAPHIC
TXT WR CHAR

85: SCR INITIALISE #BBFF

Initialize the Screen Pack.

Action:

Full initialization of the Screen Pack (as used during EMS). All Screen Pack variables and indirections are initialized, also the screen mode and the inks are initialized to their default settings.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The screen indirections (SCR READ, SCR WRITE and SCR MODE CLEAR) are set to their default routines.

The inks are set to their default colours (see Appendix V).

The ink flashing periods are set to their default values.

The screen is put into mode 1.

The screen base is set to put the screen memory at #C000..#FFFF (under the upper ROM.

The screen offset is set to 0.

The screen is cleared to ink 0.

The Graphics VDU write mode is set to FORCE mode.

The ink flashing frame flyback event is set up.

The initialization is performed in an order that attempts to avoid the previous contents of the screen becoming visible (at EMS the contents will be random).

Related entries:

GRA INITIALISE
SCR RESET
TXT INITIALISE

86: SCR RESET #BC02

Reset the Screen Pack.

Action:

Reinitializes the Screen Pack indirections and the ink colours. Also reinitializes the flash rate and Graphics VDU write mode.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The screen indirections (SCR READ, SCR WRITE and SCR MODE CLEAR) are set to their default routines.

The inks are set to their default colours (see Appendix V).

The ink flashing periods are set to their default values.

The Graphics VDU write mode is set to FORCE mode.

The inks are not passed to the hardware. This will be done when the inks flash next.

Related entries:

SCR INITIALISE
SCR SET ACCESS
SCR SET FLASHING
SCR SET INK

87: SCR SET OFFSET #BC05

Set the offset of the start of the screen.

Action:

Set the offset of the first character on the screen. By changing this offset the screen can be rolled.

Entry conditions:

HL contains the required offset.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The offset passed is masked with #07FE to make sure it is not too big to make sure that the offset is even. (The screen is only capable of rolling in two byte increments).

The screen base and screen offset are combined into a single value and sent to the hardware together.

The screen offset is used by SCR CHAR POSITION and SCR DOT POSITION to calculate screen addresses. If the screen offset is changed merely by calling the Machine Pack routine MC SCREEN OFFSET then the Text and Graphics VDUs will use incorrect screen addresses.

The offset is set to zero when the screen mode is set or the screen is cleared by calling SCR CLEAR.

Related entries:

MC SCREEN OFFSET
SCR GET LOCATION
SCR HW ROLL
SCR SET BASE
SCR SET POSITION

88: SCR SET BASE #BC08

Set the area of RAM to use for the screen memory.

Action:

Set the base address of the screen memory. This can be used to move the screen out from underneath the upper ROM or to display a prepared screen instantly.

Entry conditions:

A contains the more significant byte of the base address.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The screen memory can only be located on a 16K boundary so the value passed in masked with #C0. The default screen base, set at EMS, is #C0.

The screen offset is combined with the screen base into a single value which is sent to the hardware.

The screen base address is used by SCR CHAR POSITION and SCR DOT POSITION to calculate screen addresses. If the screen base is changed merely by calling the Machine Pack routine MC SCREEN OFFSET then the text and graphics VDUs will use incorrect screen addresses.

The screen memory is not cleared when the screen base is set, use SCR CLEAR to do this.

Related entries:

MC SCREEN OFFSET
SCR GET LOCATION
SCR SET OFFSET
SCR SET POSITION

89: SCR GET LOCATION #BC0B

Fetch current base and offset settings.

Action:

Ask where the screen memory is located and where the start of the screen is.

Entry conditions:

No conditions.

Exit conditions:

A contains the more significant byte of the base address.
HL contains the current offset.

Flags corrupt.
All other registers preserved.

Notes:

The base and offsets returned by this routine may not be the same as those set using SCR SET BASE or SCR SET OFFSET. This is because the values are masked to make them legal and the screen offset is also changed when the hardware screen rolling routine, SCR HW ROLL, is used.

Related entries:

SCR SET BASE
SCR SET OFFSET
SCR SET POSITION

90: SCR SET MODE #BC0E

Set screen into a new mode.

Action:

Put the screen into a new mode and make sure that the Text and Graphics VDUs are set up correctly.

Entry conditions:

A contains the required mode.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The mode requested is masked with #03. If the resulting value is 3 then no action is taken. Otherwise one of the following screen modes is set up:

Mode 0:	160 x 200 pixels,	20 x 25 characters.
Mode 1:	320 x 200 pixels,	40 x 25 characters.
Mode 2:	640 x 200 pixels,	80 x 25 characters.

At an early stage the screen is cleared to avoid the old contents of the screen being displayed in the wrong mode. The screen is cleared by calling the SCR MODE CLEAR indirection.

All the text and graphics windows are set to cover the whole screen and the graphics user origin is set to the bottom left corner of the screen. The cursor blobs for all text streams are turned off. Stream zero is selected.

The current text and graphics pen and paper inks are masked as appropriate for the new mode (see TXT SET PEN et al). When changing mode to a mode that allows fewer inks on the screen this may cause the pen and paper inks to change.

Related entries:

MC SET MODE
SCR GET MODE

91: SCR GET MODE #BC11

Ask the current screen mode.

Action:

Fetch and test the current screen mode.

Entry conditions:

No conditions.

Exit conditions:

If current mode is mode 0:

Carry true.
Zero false.
A contains 0.

If current mode is mode 1:

Carry false.
Zero true.
A contains 1.

If current mode is mode 2:

Carry false.
Zero false.
A contains 2.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The modes are:

Mode 0:	160 x 200 pixels,	20 x 25 characters.
Mode 1:	320 x 200 pixels,	40 x 25 characters.
Mode 2:	640 x 200 pixels,	80 x 25 characters.

Related entries:

SCR SET MODE

92: SCR CLEAR #BC14

Clear the screen (to ink zero).

Action:

Clear the whole of screen memory to zero.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

At an early stage the ink flashing is turned off and the inks are all set to the same colour as ink 0. This makes the screen clearing appear instantaneous. When all the screen memory has been set to 0 the ink flashing is turned back on (an ink flashing event is added to the frame flyback queue) and all inks are set to their proper colours.

If the text paper ink and graphics paper ink are not set to ink 0 this will become apparent on the screen when characters are written or windows are cleared.

The screen offset is set to zero.

Related entries:

GRA CLEAR WINDOW
SCR MODE CLEAR
TXT CLEAR WINDOW

93: SCR CHAR LIMITS #BC17

Ask the size of the screen in characters.

Action:

Get the last character row and column on the screen in the current mode.

Entry conditions:

No conditions.

Exit conditions:

B contains the physical last column on the screen.
C contains the physical last row on the screen.

AF corrupt.
All other registers preserved.

Notes:

The screen edges are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. This means that the last column on the screen is 19 in mode 0, 39 in mode 1 and 79 in mode 2. The last row on the screen is 24 in all modes.

Related entries:

SCR GET MODE

94: SCR CHAR POSITION #BC1A

Convert physical coordinates to a screen position.

Action:

Calculate the screen address of the top left corner of a character position on the screen. Also returns the width of a character in the current mode.

Entry conditions:

H contains the physical character column.

L contains the physical character row.

Exit conditions:

HL contains the screen address of the top left corner of the character.

B contains the width in bytes of a character in screen memory.

AF corrupt.

All other registers preserved.

Notes:

The character position is given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen.

The character position given is not checked for being legal. An illegal position (one outside the limits of the screen) will generate a meaningless screen address.

The conversion to screen address uses the following formula:

Screen address = Screen base + (Block offset MOD #0800)

where:

Block offset = (Row * 80) + (Column * Width) + Screen offset.

and:

Screen base is the address of the start of screen memory.

Width is the width of a character in bytes in the current mode (4 in mode 0, 2 in mode 1, 1 in mode 2)

Screen offset is offset of the first byte to be displayed on the screen.

Related entries:

SCR DOT POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV BYTE
SCR PREV LINE

95: SCR DOT POSITION #BC1D

Convert base coordinates to a screen position.

Action:

Calculate the screen address and mask for a pixel. Also return an indication of the number of pixels in a screen byte in the current mode.

Entry conditions:

DE contains the base X coordinate of a pixel.
HL contains the base Y coordinate of a pixel.

Exit conditions:

HL contains the screen address of the pixel.
C contains the mask for the pixel.
B contains one less than the number of pixels in a byte.

AF and DE corrupt.
All other registers preserved.

Notes:

The pixel position is given in base coordinates. i.e. (0,0) is the pixel in the bottom left corner of the screen and each coordinate position refers to a single pixel.

The pixel position is not checked for being legal (within the limits of the screen). If it is not then the screen address calculated is meaningless.

The conversion to screen address uses the following formula:

Screen address=Screen base+(Line in row*#0800)+(Row offset MOD #0800)

Where:

Screen base	is the address of the start of screen memory.
Line in row	= (199 - Y coordinate) MOD 8
Row offset	= (Row number * 80) + Byte in row + Screen offset

and:

Row number	= (199 - Y coordinate)/8
Byte in row	= X coordinate/Byte width
Screen offset	is the offset of the first byte to be displayed on the screen.

Byte width — is the number of pixels in a byte in the current mode (2 in mode 0, 4 in mode 1, 8 in mode 2).

X coordinate MOD Byte width is used to calculate the mask for the appropriate pixel.

Related entries:

GRA FROM USER
SCR CHAR POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV BYTE
SCR PREV LINE

96: SCR NEXT BYTE #BC20

Step a screen address right one byte.

Action:

Calculate the screen address of the byte right of the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the end of the screen line is not prevented. It will simply point the screen address at the next byte in the screen block. Normally this will be the first byte on a screen line 8 screen lines down from the old line (i.e. down one character row). However, moving right off the end of the last screen line in a block will point to the screen address at the start of the 48 bytes in the block that are not displayed on the screen.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT LINE
SCR PREV BYTE
SCR PREV LINE

97: SCR PREV BYTE #BC23

Step a screen address left one byte.

Action:

Calculate the screen address of the byte left of the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the start of the screen line is not prevented. It will simply point the screen address at the previous byte in the screen block. Normally this will be the last byte on a screen line 8 screen lines up from the old line (i.e. up one character row). However, moving left off the start of the top screen line in a block will point to the screen address at the last of the 48 bytes in the block that are not displayed on the screen.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV LINE

98: SCR NEXT LINE #BC26

Step a screen address down one line.

Action:

Calculate the screen address of the byte below the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the bottom of the screen is not prevented (and not recommended). After moving off the bottom the screen address is not useful.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT BYTE
SCR PREV BYTE
SCR PREV LINE

99: SCR PREV LINE #BC29

Step a screen address up one line.

Action:

Calculate the screen address of the byte above the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the top of the screen is not prevented (and not recommended). After moving off the top the screen address is not useful.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV BYTE

100: SCR INK ENCODE #BC2C

Encode an ink to cover all pixels in a byte.

Action:

Convert an ink to the encoded form that will set all pixels in a byte to the ink. This encoded ink can then be masked to generate the appropriate value to set a single pixel to the ink.

Entry conditions:

A contains an ink number.

Exit conditions:

A contains the encoded ink.

Flags corrupt.
All other registers preserved.

Notes:

The encoding is not trivial as the pixels in a byte are interleaved and also the bits in a pixel are not in the obvious order. The pixel bits are (most significant to least significant):

	Mode 0	Mode 1	Mode 2
Leftmost pixel:	Bits 1,5,3,7	Bits 3,7	Bit 7
			Bit 6
		Bits 2,6	Bit 5
			Bit 4
	Bits 0,4,2,6	Bits 1,5	Bit 3
			Bit 2
		Bits 0,4	Bit 1
Rightmost pixel:			Bit 0

The Text and Graphics VDUs store their pen and paper inks in this encoded form for ease of use internally. This saves time converting the ink for each pixel plotted.

The encoding is different in different modes and so all inks have to be re-encoded when the screen mode is changed. SCR SET MODE does this automatically for the Text VDU and Graphics VDU pen and paper inks.

Related entries:

SCR INK DECODE

101: SCR INK DECODE #BC2F

Decode an encoded ink.

Action:

Convert an encoded ink to the appropriate ink number.

Entry conditions:

A contains an encoded ink.

Exit conditions:

A contains the ink number.

Flags corrupt.
All other registers preserved.

Notes:

The decoding is performed by decoding the ink of the leftmost pixel in the encoded ink. The ink for this pixel is encoded in the following bits (most significant to least significant) in the various screen modes:

Mode 0:	Bits 1,5,3,7
Mode 1:	Bits 3,7
Mode 2:	Bit 7

Related entries:

SCR INK ENCODE

102: SCR SET INK #BC32

Set the colours in which to display an ink.

Action:

Set which two colours will be used to display an ink. If the two colours are the same then the ink will remain a steady colour. If the colours are different then the ink will alternate between these two colours.

Entry conditions:

A contains an ink number.

B contains the first colour.

C contains the second colour.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The ink number is masked with #0F to make sure it is legal, and the colours are masked with #1F. Colours 27..31 are not intended for use; they are merely duplicates of other colours available.

The new colours for an ink are not sent to the hardware immediately. They are stored and will appear on the screen when the next frame flyback occurs.

The length of time for which each colour is displayed on the screen can be set by calling SCR SET FLASHING.

The inks are set to their default colours at EMS and when SCR RESET is called.

The various colours available and the default ink colours set are described in Appendix V.

Related entries:

GRA SET PAPER
GRA SET PEN
SCR GET INK
SCR SET BORDER
SCR SET FLASHING
TXT SET PAPER
TXT SET PEN

103: SCR GET INK #BC35

Ask the colours an ink is currently displayed in.

Action:

Get the two colours that are used to display an ink on the screen.

Entry conditions:

A contains an ink number.

Exit conditions:

B contains the first colour.
C contains the second colour.

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The ink number is masked with #0F to make sure it is legal. The colours returned may not be the same as those supplied to the Screen Pack as the colours are masked when they are set.

The new colours for an ink are not sent to the hardware immediately when they are set. They are stored and appear on the screen when the next frame flyback occurs. This means that the colours returned may not actually be visible to the user yet.

The default settings for the inks and the various colours available are described in Appendix V.

Related entries:

GRA GET PAPER
GRA GET PEN
SCR GET BORDER
SCR SET INK
TXT GET PAPER
TXT GET PEN

104: SCR SET BORDER #BC38

Set the colours in which to display the border.

Action:

Set which two colours will be used to display the border. If the two colours are the same then the border will remain a steady colour. If the colours are different then the border will alternate between these two colours.

Entry conditions:

B contains the first colour.

C contains the second colour.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The colours are masked with #1F to ensure that they are legal. Colours 27..31 are not intended for use; they are merely duplicates of other colours available.

The new colours for the border are not sent to the hardware immediately. They are stored and will appear on the screen when the next frame flyback occurs.

The length of time for which each colour is displayed on the screen can be set by calling SCR SET FLASHING.

The border is set to its default colours at EMS and when SCR RESET is called. The default colour and the colours available are described in Appendix V.

Related entries:

SCR GET BORDER
SCR SET FLASHING
SCR SET INK

105: SCR GET BORDER #BC3B

Ask the colours the border is currently displayed in.

Action:

Get the two colours used to display the border on the current screen.

Entry conditions:

No conditions.

Exit conditions:

B contains the first colour.
C contains the second colour.

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The colours returned may not be the same as those supplied to the Screen Pack as they are masked when they are set.

The new colours for the border are not sent to the hardware immediately when they are set. They are stored and appear on the screen when the next frame flyback occurs. This means that the colours returned may not actually be visible to the user yet.

The default border colour and the colours available are described in Appendix V.

Related entries:

SCR GET INK
SCR SET BORDER

106: SCR SET FLASHING #BC3E

Set the flash periods.

Action:

Set for how long each of the two colours for the inks and the border are to be displayed on the screen. These settings apply to all inks and the border.

Entry conditions:

H contains the period for the first colour.

L contains the period for the second colour.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

The flash periods are given in frame flybacks (1/50 or 1/60 of a second). A period of 0 is taken to mean a period of 256.

The default setting for the flash periods is 10 frame flybacks (1/5 or 1/6 of a second). This is set at EMS and when SCR RESET is called.

The new flash periods are not used immediately but when the inks next flash.

Related entries:

SCR GET FLASHING
SCR SET BORDER
SCR SET INK

107: SCR GET FLASHING #BC41

Ask the current flash periods.

Action:

Get the time for which each of the two colours associated with an ink or the border is displayed.

Entry conditions:

No conditions.

Exit conditions:

H contains the period for the first colour.
L contains the period for the second colour.

AF corrupt.
All other registers preserved.

Notes:

The flash periods are given in frame flybacks (1/50 or 1/60 of a second).

A period of 0 means 256.

Related entries:

SCR SET FLASHING

108: SCR FILL BOX #BC44

Fill a character area of the screen with an ink.

Action:

Fill a rectangular area of the screen with an ink. The boundaries of this area are given in character positions.

Entry conditions:

A contains the encoded ink to fill the area with.

H contains the physical left column of the area to fill.

D contains the physical right column of the area to fill.

L contains the physical top row of the area to fill.

E contains the physical bottom row of the area to fill.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The area boundaries are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. They are not checked for legality. If illegal boundaries are passed (edges off the screen) then unpredictable effects may occur.

The screen is written directly without using any other write routine. The current Graphics VDU write mode is therefore ignored.

Related entries:

SCR CLEAR
SCR FLOOD BOX
TXT CLEAR WINDOW

109: SCR FLOOD BOX #BC47

Fill a byte area of the screen.

Action:

Fill a rectangular area of the screen with an ink. The boundaries of the area must lie on byte boundaries. This routine will not fill an arbitrary area of the screen to pixel a boundary.

Entry conditions:

C contains the encoded ink to fill the area with.

HL contains the screen address of the top left corner of the area to fill.

D contains the (unsigned) width of the area to fill in bytes.

E contains the (unsigned) height of the area to fill in screen lines.

Exit conditions:

AF, BC, DE and HL corrupt.

All other register preserved.

Notes:

The whole of the rectangle being cleared must lie on the screen. If any of it lies off the screen then unpredictable effects may occur.

A height or width of 0 is taken to mean 256 (which is too large to fit on the screen).

The screen is written directly without using any other write routine. The current Graphics VDU write mode is therefore ignored.

Related entries:

GRA CLEAR WINDOW
SCR CLEAR
SCR FILL BOX

110: SCR CHAR INVERT #BC4A

Invert a character position.

Action:

All pixels at a character position that are written in one ink are rewritten in a second ink, and vice versa. This gives an inverse effect to the character position. Inverting the character a second time will restore the original inks. This effect is used to draw the Text VDU cursors.

Entry conditions:

B contains an encoded ink.

C contains another encoded ink.

H contains a physical character column.

L contains a physical character row.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The character position is given in physical coordinates i.e. Row 0, column 0 is the top left corner of the screen.

The character position given is not checked for being legal. An illegal position (one outside the limits of the screen) will have unpredictable effects.

All pixels at the character position are exclusive-ored with the exclusive-or of the two inks supplied. Pixels at the character position that are set to one of the two inks supplied will therefore be set to the other supplied ink. Pixels set to other inks will also be altered.

Related entries:

TXT PLACE CURSOR
TXT REMOVE CURSOR

111: SCR HW ROLL #BC4D

Move the whole screen up or down eight pixel lines (one character).

Action:

Roll the screen using the hardware. The new line appearing on the screen is cleared.

Entry conditions:

If the screen is rolled down:
  B must be zero.

If the screen is to roll up:
  B must be non-zero.

Always:
  A contains the encoded ink to clear the new line to.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The screen is rolled by changing the screen offset (see SCR SET OFFSET).

Rolling the screen upwards moves the screen contents up and clears the new bottom line. The screen offset is therefore increased by 80 (MOD #0800).

Rolling the screen downwards moves the screen contents down and clears the new top line. The screen offset is therefore decreased by 80 (MOD #0800).

The new line is cleared by writing to it directly thus the Graphics VDU write mode is ignored.

The Text VDU roll count is not changed by this routine (see TXT GET WINDOW).

Special precautions are taken to make sure that the screen is kept looking presentable during the rolling and in particular during the clearing of the new line. Principally this consists of clearing the new line in two parts. First the part that is not visible on the screen (by virtue of the screen addressing) is cleared. Then, after waiting for frame flyback and changing the screen offset, the second half of the line that was part of the line that just rolled off the screen is cleared.

Related entries:

SCR SET OFFSET
SCR SW ROLL

112: SCR SW ROLL #BC50

Move an area of the screen up or down eight pixel lines (one character).

Action:

Roll an area of the screen by copying. The area to be rolled is specified in character positions.

Entry conditions:

If the screen is to roll down:

B must be zero.

If the screen is to roll up:

B must be non-zero.

Always:

A contains the encoded ink to clear the new line to.
H contains the physical left column of the area to roll.
D contains the physical right column of the area to roll.
L contains the physical top row of the area to roll.
E contains the physical bottom row of the area to roll.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The area boundaries are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. The boundaries are not checked for legality. If illegal boundaries are passed (edges off the screen) then unpredictable effects may occur.

Rolling the area upwards moves the contents up and clears the new bottom line. Rolling the area downwards moves the area contents down and clears the new top new line.

The line is cleared by writing to it directly; the Graphics VDU write mode is ignored.

The Text VDU roll count is not changed by this routine (see TXT GET WINDOW).

Special precautions are taken to make sure that the screen is kept looking presentable during the rolling. Principally this consists of waiting for frame flyback before performing the copy.

Related entries:

SCR HW ROLL

113: SCR UNPACK #BC53

Expand a character matrix for the current screen mode.

Action:

Convert a matrix from its standard form to a set of pixel masks as appropriate for the current screen mode.

Entry conditions:

HL contains the address of a matrix.

DE contains the address of an area to unpack into.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The matrix is converted into a series of masks which cover all the screen bytes in the character. This means that each byte of the matrix is converted to 4 bytes in mode 0, 2 bytes in mode 1 and 1 byte in mode 2. Thus the unpacking area must be 32, 16 or 8 bytes long.

If a bit in the matrix is set then the appropriate pixel mask is included in the unpacked version (the bits are set to one). Otherwise the pixel mask is not included in the unpacked version (the bits are set to zero).

Related entries:

SCR REPACK

114: SCR REPACK #BC56

Compress a character matrix to the standard form.

Action:

A character on the screen is converted to a matrix by comparing each pixel with an ink. If the pixel is set to that ink then the appropriate bit in the character matrix is set, otherwise the bit is cleared.

Entry conditions:

A contains the encoded ink to match against.

H contains the physical character column to read from.

L contains the physical character row to read from.

DE contains the address of the area to construct the matrix in.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The character position is given in physical coordinates in which row 0, column 0 is the top left corner of the screen.

The character position given is not checked for legality. An illegal position (one outside the limits of the screen) will have unpredictable effects.

The matrix produced has the normal layout. It is 8 bytes long, stored top line first and bottom line last, the most significant bit of the byte refers to the leftmost pixel of a line and the least significant bit to the rightmost pixel.

Because the pixels are tested for being set to only one ink the matrix produced is not an exact representation of what is in the screen. It may be necessary, when trying to read characters from the screen, to repack using various different inks.

Related entries:

SCR UNPACK
TXT RD CHAR

115: SCR ACCESS #BC59

Set the screen write mode for the Graphics VDU.

Action:

Set the Graphics VDU write mode so that the Graphics VDU plots pixels by writing, anding, oring or exclusive-oring.

Entry conditions:

A contains the required write mode.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The write mode is masked with #03 to make it legal. The write modes are:

0: FORCE mode:	NEW = INK
1: XOR mode:	NEW = INK exclusive-or OLD
2: AND mode:	NEW = INK and OLD
3: OR mode:	NEW = INK or OLD

NEW is the final setting of the pixel.
OLD is the current setting of the pixel.
INK is the ink being plotted.

The default mode is FORCE mode (mode 0) and is set at EMS and when SCR RESET is called.

Setting the write mode affects how the indirection routine SCR WRITE sets pixels. Graphics VDU plotting routines call this indirection to set pixels and so the write mode affects the Graphics VDU. No Text VDU routines call this indirection (they set pixels on the screen directly) and so the write mode does not affect the Text VDU. The routines that clear areas of the screen (e.g. GRA CLEAR WINDOW) act like the Text VDU and are unaffected by the write mode.

Related entries:

GRA DEFAULT
SCR INITIALISE
SCR RESET
SCR WRITE

116: SCR PIXELS #BC5C

Write a pixel to the screen ignoring the Graphics VDU write mode.

Action:

Write a pixel or pixels to the screen. The position to write at is given by a screen address and pixel mask. The pixel is always set to the ink supplied whatever mode of writing the Graphics VDU is using.

Entry conditions:

B contains the encoded ink to write.

C contains the mask for the pixel(s).

HL contains the screen address of the pixel(s).

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The screen address is not checked and so passing an invalid screen address will have unpredictable results.

The pixel mask may be a combined mask for more than one pixel (thus speeding up plotting in certain cases).

To plot a pixel using the Graphics VDU write mode SCR WRITE should be called. SCR PIXELS is equivalent to calling SCR WRITE when the default mode (FORCE mode) is selected. The Text VDU sets the pixels in characters using FORCE mode.

Related entries:

SCR WRITE

117: SCR HORIZONTAL #BC5F

Plot a purely horizontal line.

Action:

Draw a line on the screen that runs horizontally. The pixels on the line are plotted using the SCR WRITE indirection and thus use the current Graphics VDU write mode.

Entry conditions:

A contains the encoded ink to draw in.

DE contains the base X coordinate of the start of the line.

BC contains the base X coordinate of the end of the line.

HL contains the base Y coordinate of the line.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The endpoints of the line are given in base coordinates. i.e. (0,0) is the pixel in the bottom left corner of the screen and each coordinate position refers to a single pixel.

The endpoints are not checked for being legal (within the limits of the screen). If they are not legal then unpredictable effects may occur.

The start X coordinate must be less than or equal to the end X coordinate.

This routine may be used to duplicate the method that the Graphics VDU uses for plotting lines - it splits a line that is more horizontal than vertical into a number of segments that are purely horizontal and plots these separately.

Related entries:

GRA FROM USER
GRA LINE ABSOLUTE
GRA LINE RELATIVE
SCR VERTICAL

118: SCR VERTICAL #BC62

Plot a purely vertical line.

Action:

Draw a line on the screen that runs vertically. The SCR WRITE indirection is used to plot pixel in the line thus the current Graphics VDU write mode is used.

Entry conditions:

A contains the encoded ink to draw in.

DE contains the base X coordinate of the line.

HL contains the base Y coordinate of the start of the line.

BC contains the base Y coordinate of the end of the line.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The endpoints of the line are given in base coordinates. i.e. (0,0) is the pixel in the bottom left corner of the screen and each coordinate position refers to a single pixel.

The endpoints are not checked for being legal (within the limits of the screen). If they are not legal then unpredictable effects may occur.

The start Y coordinate must be less than or equal to the end Y coordinate.

This routine may be used to duplicate the method that the Graphics VDU uses for plotting lines - it splits a line that is more vertical than horizontal into a number of segments that are purely vertical and plots these separately.

Related entries:

GRA FROM USER
GRA LINE ABSOLUTE
GRA LINE RELATIVE
SCR HORIZONTAL

119: CAS INITIALISE #BC65

Initialize the Cassette Manager.

Action:

Full initialization of the Cassette Manager (as used during EMS).

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Operations carried out are:

All streams are marked closed.
The default write speed is set up.
The prompt messages are turned off.
The cassette motor is turned off (except on V1.0 firmware).

Related entries:

CAS IN ABANDON
CAS NOISY
CAS OUT ABANDON
CAS SET SPEED
CAS STOP MOTOR

120: CAS SET SPEED #BC68

Set the write speed.

Action:

Set the length to write bits and the amount of write precompensation to apply.

Entry conditions:

HL contains the length of half a zero bit.
A contains the precompensation to apply.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The speed supplied is the length of half a zero bit in microseconds. A one bit is written as twice the length of a zero bit. The speed supplied can be related to the average baud rate (assuming equal numbers of ones and zeros) by the following equation:

Average baud rate

= 1 000 000 / (3 * Halfzero length)
= 333 333 / Halfzero length

The halfzero length must lie between 130 and 480 microseconds. Values outside this range will cause read and write errors.

The precompensation supplied is the extra length, in microseconds, to add to half a one bit and to subtract from half a zero bit under certain conditions. The amount of precompensation required varies with the speed (more is required at higher baud rates).

The precompensation may lie between 0 and 255 microseconds although the higher settings are not useful as they will cause read and write errors.

The default half zero length and precompensation settings are 333 microseconds (1000 baud) and 25 microseconds respectively. The commonly used faster setting is 167 microseconds (2000 baud) with 50 microseconds of precompensation. These values have been determined after extensive testing and the user is advised to stick to them.

Related entries:

CAS INITIALISE

121: CAS NOISY #BC6B

Enable or disable prompt messages.

Action:

Disabling messages will prevent the prompt and information messages from being printed. It will not prevent error messages from being printed. Enabling messages allows all messages to be printed.

Entry conditions:

If messages are to be enabled:

A must be zero.

If messages are to be disabled:

A must be non-zero.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The prompt and information messages which are turned off are:

Press PLAY then any key:
Press REC and PLAY then any key:
Found <FILENAME> block <N>
Loading <FILENAME> block <N>
Saving <FILENAME> block <N>

The error messages which are not turned off are:

Read error <x>
Write error a
Rewind tape

Related entries:

CAS INITIALISE

122: CAS START MOTOR #BC6E

Start the cassette motor.

Action:

Turn the cassette motor on and wait for it to pick up speed if it was previously off.

Entry conditions:

No conditions.

Exit conditions:

If the motor turned on OK:

Carry true.

If the user hit escape:

Carry false.

Always:

A contains the previous motor state.

Other flags corrupt.
All other registers preserved.

Notes:

If the motor is not already on then the routine waits for approximately two seconds to allow the tape to reach full speed.

The motor is always turned on by this routine. If the user hits the escape key then the time spent waiting for the motor to pick up speed is truncated.

The previous motor state may be passed to CAS RESTORE MOTOR.

Related entries:

CAS RESTORE MOTOR
CAS STOP MOTOR

123: CAS STOP MOTOR #BC71

Stop the cassette motor.

Action:

Turn the cassette motor off and return its previous state.

Entry conditions:

No conditions.

Exit conditions:

If the motor was turned off OK:

Carry true.

If the user hit escape:

Carry false.

Always:

A contains the previous motor state.

Other flags corrupt.
All other registers preserved.

Notes:

The motor is always turned off by this routine. There is no delay to allow the motor to slow down.

The previous motor state may be passed to CAS RESTORE MOTOR.

Related entries:

CAS RESTORE MOTOR
CAS START MOTOR

124: CAS RESTORE MOTOR #BC74

Restore previous state of cassette motor.

Action:

Turn the cassette motor on or off again. Wait for motor to pick up speed when turning the motor on if it is currently off.

Entry conditions:

A contains the previous motor state.

Exit conditions:

If the motor was turned on or off OK:

Carry true.

If the user hit escape:

Carry false.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

This routine uses the previous motor state as returned by CAS START MOTOR or CAS STOP MOTOR.

If calling this routine results in the motor being turned on when it is currently off then the routine waits for approximately two seconds to allow the tape to reach full speed.

The motor is always turned on or off (as appropriate) by this routine. If the user hits the escape key then this merely truncates the time spent waiting for the motor to pick up speed.

Related entries:

CAS START MOTOR
CAS STOP MOTOR

125: CAS IN OPEN #BC77

Open a file for input.

Action:

Set up the read stream for reading a file and read the first block.

Entry conditions:

B contains the length of the filename.

HL contains the address of the filename.

DE contains the address of a 2K buffer to use.

Exit conditions:

If the file was opened OK:

Carry true.
Zero false.
HL contains the address of a buffer containing the file header.
DE contains the data location (from the header).
BC contains the logical file length (from the header).
A contains the file type (from the header).

If the stream is in use:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.
BC,DE and HL corrupt.

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.
BC,DE and HL corrupt.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can return two error numbers:

#00:	The user hit escape.
#0E:	The stream is already in use.

The 2K buffer (2048 bytes) supplied is used to store the contents of a block of the file when it is read from tape. It will remain in use until the file is closed by calling either CAS IN CLOSE or CAS IN ABANDON. The buffer may lie anywhere in memory, even underneath a ROM.

The filename passed is copied into the read stream descriptor. If it is longer than 16 characters then it is truncated to 16 characters. If it is shorter then 16 characters then it is padded with nulls (#00) to 16 characters. While the filename may contain any character, it is best to avoid nulls. Lower case ASCII letters (characters #61..#7A) are converted to their upper case equivalents (characters #41..#5A). The filename may lie anywhere in RAM, even underneath a ROM.

The filename is normally the name of the file that is to be read. However, a zero length filename (or one starting with a null) is treated specially. It is taken to mean read the next file on the tape.

When the file is opened for reading the first block of the file is read immediately. The address of the area where the header from this block is stored is passed back to the user so that information can be extracted from it. This area will lie in the central 32K of RAM. The user is not allowed to write to the header, only read from it. The Cassette Manager uses fields in the header for its own purposes and so these may differ from those read from the tape. The file type, logical length, entry point and all user fields will remain unchanged. (see section 8 for a description of the header).

Related entries:

CAS IN ABANDON
CAS IN CHAR
CAS IN CLOSE
CAS IN DIRECT
CAS IN OPEN (DISC)
CAS OUT OPEN

125: CAS IN OPEN (DISC) #BC77

Open a file for input.

Action:

Set up the read stream for reading a file and read the header if there is one, other wise create a fake header in store.

Entry conditions:

B contains the length of the filename.

HL contains the address of the filename.

DE contains the address of a 2K buffer to use.

Exit conditions:

If the file was opened OK:

Carry true.
Zero false.
HL contains the address of a buffer containing the file header.
DE contains the data location (from the header).
BC contains the logical file length (from the header).
A contains the file type (from the header).

If the stream is already open:

Carry false.
Zero false.
A contains an error number (#0E).
BC,DE and HL corrupt.

If the open failed for any other reason:

Carry false.
Zero true.
A contains an error number.
BC,DE and HL corrupt.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

The 2K buffer (2048 bytes) supplied is used to store the contents of a block of the file when it is read from disc. It will remain in use until the file is closed by calling either CAS IN CLOSE or CAS IN ABANDON. The buffer may lie anywhere in memory, even underneath a ROM.

The filename must conform to the AMSDOS conventions with no wild cards. The filename may lie anywhere in RAM, even underneath a ROM.

If the type part of the filename is omitted AMSDOS will attempt to open, in turn, a file with the following type parts '.', '.BAS', '.BIN'. If none of these exist then the open will fail.

When the file is opened the first record of the file is read immediately. If this record contains a header then it is copied into store, otherwise a fake header is constructed in store. The address of the area where the header is stored is passed back to the user so that information can be extracted from it. This area will lie in the central 32K of RAM. The user is not allowed to write to the header, only read from it. AMSDOS uses fields in the header for its own purposes and so these may differ from those read from the disc. The file type, logical length, entry point and all user fields will remain unchanged.

Related entries:

CAS IN ABANDON (DISC)
CAS IN CHAR (DISC)
CAS IN CLOSE (DISC)
CAS IN DIRECT (DISC)
CAS IN OPEN
CAS OUT OPEN (DISC)

126: CAS IN CLOSE #BC7A

Close the input file properly.

Action:

Mark the read stream as closed.

Entry conditions:

No conditions.

Exit conditions:

If the stream was closed OK:

Carry true.
A corrupt.

If the stream was not open:

Carry false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.

Always:

BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

This routine can only return one error number:

#0E: The stream is not open

This routine should be called to close a file after reading from it using either CAS IN CHAR or CAS IN DIRECT.

The user may reclaim the buffer passed to CAS IN OPEN after calling this routine.

Related entries:

CAS IN ABANDON
CAS IN CLOSE (DISC)
CAS IN OPEN
CAS OUT CLOSE

126: CAS IN CLOSE (DISC) #BC7A

Close the input file properly.

Action:

Mark the read stream as closed.

Entry conditions:

No conditions.

Exit conditions:

If the stream was closed OK:

Carry true.
Zero false.
A corrupt.

If the stream is not open:

Carry false.
Zero false.
A contains an error number (#0E).

If the close failed for any other reason:

Carry false.
Zero true.
A contains an error number.

Always:

BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

This routine should be called to close a file after reading from it using either CAS IN CHAR or CAS IN DIRECT.

The user may reclaim the buffer passed to CAS IN OPEN after calling this routine.

The drive motor is turned off immediately after the input file has closed. This is done so that a loaded program which takes over the machine is not left with the motor running indefinitely.

Related entries:

CAS IN ABANDON (DISC)
CAS IN CLOSE
CAS IN OPEN (DISC)
CAS OUT CLOSE (DISC)

127: CAS IN ABANDON #BC7D

Close the input file immediately.

Action:

Abandon reading from the read stream and close it.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine is intended for use after an error or in similar circumstances.

The user may reclaim the buffer passed to CAS IN OPEN after calling this routine.

Related entries:

CAS IN ABANDON (DISC)
CAS IN CLOSE
CAS IN OPEN
CAS OUT ABANDON

127: CAS IN ABANDON (DISC) #BC7D

Close the input file immediately.

Action:

Abandon reading from the read stream and close it.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine is intended for use after an error or in similar circumstances.

The user may reclaim the buffer passed to CAS IN OPEN after calling this routine.

Related entries:

CAS IN ABANDON
CAS IN CLOSE (DISC)
CAS IN OPEN (DISC)
CAS OUT ABANDON (DISC)

128: CAS IN CHAR #BC80

Read a character from the input file.

Action:

Read a character from the input stream. Fetches blocks from tape as required.

Entry conditions:

No conditions.

Exit conditions:

If the character was read OK:

Carry true.
Zero false.
A contains the character read from the file.

If the end of the file was found:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E or #0F).
In V1.0: A corrupt.

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can return three error numbers:

#00:	The user hit escape.
#0E:	The stream is not open for reading characters or the user hit escape previously.
#0F:	Have reached the end of the file.

Once the first character has been read from a file it can only be used for character by character access. It is not possible to switch to direct reading (by CAS IN DIRECT).

Related entries:

CAS IN CHAR (DISC)
CAS IN CLOSE
CAS IN DIRECT
CAS IN OPEN
CAS OUT CHAR
CAS RETURN
CAS TEST EOF

128: CAS IN CHAR (DISC) #BC80

Read a character from an input file.

Action:

Read a character from the input stream.

Entry conditions:

No conditions.

Exit conditions:

If the character was read OK:

Carry true.
Zero false.
A contains the character read from the file.

If the end of the file was found, or stream not open as expected:

Carry false.
Zero false.
A contains an error number (#0E, #0F or #1A).

If failed for any other reason:

Carry false.
Zero true.
A contains an error number.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

Once the character has been read from a file the rest of the file may only be read character by character (using CAS IN CHAR). It is impossible to switch to direct reading (by CAS IN DIRECT).

The CP/M end of file character (#1A) is treated as end of file (carry false, zero false). However, it is possible to continue reading characters until the hard end of file. The error number returned is set to #1A for soft (CP/M) end of file and #0F for hard end of file. The action for spotting soft and of file is not performed by the equivalent cassette version of the routine and will never return #1A when carry is false.

If a file containing binary data is read using this routine then it will be necessary to spot soft EOF and ignore it.

Related entries:

CAS IN CHAR
CAS IN CLOSE (DISC)
CAS IN DIRECT (DISC)
CAS IN OPEN (DISC)
CAS OUT CHAR (DISC)
CAS RETURN (DISC)
CAS TEST EOF (DISC)

129: CAS IN DIRECT #BC83

Read the input file into store.

Action:

Read the input file directly into store in one go rather than one character at a time.

Entry conditions:

HL contains the address to put the file (anywhere in RAM).

Exit conditions:

If the file was read OK:

Carry true.
Zero false.
HL contains the entry address (from the header).
A corrupt.

If the file was not open as expected:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.
HL corrupt.

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.
HL corrupt.

Always:

BC, DE, IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can return two error numbers:

#00: The user hit escape.

#0E: The stream is not open for reading directly or the user hit escape previously.

The read stream must be newly opened (by CAS IN OPEN). If the stream has been used for character access (by calling CAS IN CHAR) then it is not possible to directly read the file. Neither is it possible to directly read from the file more than once. This will merely corrupt the copy of the file read.

The buffer of data read when the stream was opened is copied to its correct position and the remainder of the file (if any) is also read into store.

Related entries:

CAS IN CHAR
CAS IN CLOSE
CAS IN DIRECT (DISC)
CAS IN OPEN
CAS OUT DIRECT

129: CAS IN DIRECT (DISC) #BC83

Read the input file into store.

Action:

Read the input file directly into store in one go rather than one character at a time.

Entry conditions:

HL contains the address to put the file (anywhere in RAM).

Exit conditions:

If the file was read OK:

Carry true.
Zero false.
HL contains the entry address (from the header).
A corrupt.

If the stream is not open as expected:

Carry false.
Zero false.
A contains an error number (#0E).
HL corrupt.

If the read failed for any other reason:

Carry false.
Zero true.
A contains an error number.
HL corrupt.

Always:

BC, DE, IX and other flags corrupt.
All other registers preserved.

Notes:

The read stream must be newly opened (by CAS IN OPEN). If the stream has been used for character access (by calling CAS IN CHAR or CAS TEST EOF) then it is not possible to directly read the file. Neither is it possible to directly read from the file more than once. (Any attempt to do so will corrupt the copy of the file read.)

If the file has a header then the number of bytes read is that recorded in the 24 bit file length field (bytes 64..66 of the disc file header). If there is no header the file is read until hard end of file.

The CP/M end of file character, #1A, is not treated as end of file.

Related entries:

CAS IN CHAR (DISC)
CAS IN CLOSE (DISC) **CAS IN OPEN (DISC)**
CAS IN DIRECT **CAS OUT DIRECT (DISC)**

130: CAS RETURN #BC86

Put the last character read back.

Action:

Put the last character read by CAS IN CHAR back into the read buffer. The character will be re-read next time CAS IN CHAR is called.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

It is only possible to use this routine to return the last character that has been read by CAS IN CHAR. At least one character must have been read since:

the stream was opened
or the last character was returned
or the last test for end of file was made.

Related entries:

CAS IN CHAR
CAS RETURN (DISC)

130: CAS RETURN (DISC) #BC86

Put the last character read back.

Action:

Put the last character read by CAS IN CHAR back into the read buffer. The character will be re-read next time CAS IN CHAR is called.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

It is only possible to use this routine to return the last character that has been read by CAS IN CHAR. At least one character must have been read since:

the stream was opened
or the last character was returned
or the last test for end of file was made.

Related entries:

CAS IN CHAR (DISC)
CAS RETURN

131: CAS TEST EOF #BC89

Have we reached the end of the input file yet?

Action:

Test if the end of the input file has been reached.

Entry conditions:

No conditions.

Exit conditions:

If the end of the file was not found:

Carry true.
Zero false.
A corrupt.

If the end of the file was found:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E or #0F).
In V1.0: A corrupt.

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can return three error numbers:

#00: The user hit escape.

#0E: The stream is not open for reading characters or the user hit escape previously.

#0F: Have reached the end of the file.

Calling this routine put the stream into character input mode. It is not possible to use direct reading after calling this routine.

It is not possible to call CAS RETURN after this routine has been called. A character must be read first.

Related entries:

CAS IN CHAR
CAS TEST EOF (DISC)

131: CAS TEST EOF (DISC) #BC89

Have we reached the end of the input file yet?

Action:

Test if the end of the input file has been reached.

Entry conditions:

No conditions.

Exit conditions:

If the end of the file was not found:

Carry true.
Zero false.
A corrupt.

If the end of the file was found or stream was not open as expected:

Carry false.
Zero false.
A contains an error number (#0E,#0F or #1A).

If failed for any other reason:

Carry false.
Zero true.
A contains an error number.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

This routine will report end of file if either there are no more characters in the file or if the next character to be read is the CP/M end of file character, #1A.

Calling this routine puts the stream into character input mode. It is not possible to use direct reading after calling this routine.

It is not possible to call CAS RETURN after this routine has been called. A character must be read first.

Related entries:

CAS IN CHAR (DISC)
CAS TEST EOF

132: CAS OUT OPEN #BC8C

Open a file for output.

Action:

Set up the write stream for output.

Entry conditions:

B contains the length of the filename.

HL contains the address of the filename.

DE contains the address of a 2K buffer to use.

Exit conditions:

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.
HL corrupt.

If the stream is in use already:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.
HL corrupt.

If the file was opened OK:

Carry true.
Zero false.
HL contains the address of a buffer containing the header that will be written to each file block.

Always:

BC,DE,IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can only return two error numbers.

#00: The user hit escape.

#0E: The stream is already open.

When writing files character by character the 2K buffer (2048 bytes) supplied is used to store the contents of a block of the file before it is written to tape. It will remain in use until the file is closed by calling either CAS OUT CLOSE or CAS OUT ABANDON. The buffer may reside anywhere in memory - even underneath a ROM.

When the stream is opened for writing, a header is set up which will be written at the start of each block of the file. Many of the fields in the header are set by the Cassette Manager but the remainder are available for use by the user. The address of this header is passed to the user so that information can be stored in it. The user may write to the file type, logical length, entry point and all user fields. The user is not allowed to write to any other field in the header. The user settable fields are all zeroized initially, with the exception of the file type which is set to unprotected ASCII version 1. (See section 8.4 for a description of the header).

The filename passed is copied into the write stream descriptor. If it is longer than 16 characters then it is truncated to 16 characters. If it is shorter than 16 characters then it is padded with nulls (#00) to 16 characters. While the filename may contain any character, it is best to avoid nulls. Lower case ASCII letters (characters #61..#7A) are converted to their upper case equivalents (characters #41..#5A). The filename may lie anywhere in RAM, even underneath a ROM.

Related entries:

CAS IN OPEN
CAS OUT ABANDON
CAS OUT CHAR
CAS OUT CLOSE
CAS OUT DIRECT
CAS OUT OPEN (DISC)

132: CAS OUT OPEN (DISC) #BC8C

Open a file for output.

Action:

Set up the write stream for output.

Entry conditions:

B contains the length of the filename.

HL contains the address of the filename.

DE contains the address of a 2K buffer to use.

Exit conditions:

If the file was opened OK:

Carry true.
Zero false.
HL contains the address of the buffer containing the header.
A corrupt.

If the stream is open already:

Carry false.
Zero false.
A contains an error number (#0E).
HL corrupt.

If the open failed for any other reason:

Carry false.
Zero true.
A contains an error number.
HL corrupt.

Always:

BC,DE,IX and other flags corrupt.
All other registers preserved.

Notes:

When characters are output to the file using CAS OUT CHAR the 2K buffer supplied is used by AMSDOS to buffer the output. It will remain in use until the file is closed by calling either CAS OUT CLOSE or CAS OUT ABANDON. The buffer may reside anywhere in memory - even underneath a ROM.

The filename passed must conform to AMSDOS conventions with no wild cards. It is copied into the write stream header. The filename my lie anywhere in RAM - even underneath a ROM.

The file is opened with a type part of '.$$$' regardless of the type part supplied. Any existing file with the same name and type part of '.$$$' is deleted. The file is renamed to its supplied name when CAS OUT CLOSE is called.

When the stream is opened aa header is set up. Many of the fields in the header are set by AMSDOS but the remainder are available for use by the user. The address of this header is passed to the user so the information can be stored in it. The user may write to the file type, logical length, entry point and all other fields. The user is not allowed to write to any other field of the header. The user settable fields are all zeroized initially, with the exception of the file type which is set to unprotected ASCII version 1.

The header type field must be written to before CAS OUT CHAR or CAS OUT DIRECT is called. The type field must not be altered after calling either of these routines. If the file type is set to any type other than unprotected ASCII then space will be preserved for the header which when the file is closed.

Related entries:

CAS IN OPEN (DISC)
CAS OUT ABANDON (DISC)
CAS OUT CHAR (DISC)
CAS OUT CLOSE (DISC)
CAS OUT DIRECT (DISC)
CAS OUT OPEN

133: CAS OUT CLOSE #BC8F

Close the output file properly.

Action:

Mark the write stream as closed and write the last buffer area of data to tape.

Entry conditions:

No conditions.

Exit conditions:

If the stream was closed OK:

Carry true.
Zero false.
A corrupt.

If the stream is not open:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can return two error numbers:

#00: The user hit escape.

#0E: The stream is not open.

It is necessary to call this routine after using CAS OUT CHAR or CAS OUT DIRECT to cause the last block of data to be written to the tape. If the block is zero bytes long (nothing has been written to the file) then nothing is written to tape.

If writing is to be abandoned then CAS OUT OPEN should be called as this does not write the last block of data to the tape.

If the user hits escape during the writing of the last block then the file is left open and is not closed.

The user may reclaim the buffer passed to CAS OUT OPEN after calling this routine.

Related entries:

CAS IN CLOSE
CAS OUT ABANDON
CAS OUT CLOSE (DISC)
CAS OUT OPEN

133: CAS OUT CLOSE (DISC) #BC8F

Close the output file properly.

Action:

Mark the write stream as closed and give it its correct name.

Entry conditions:

No conditions.

Exit conditions:

If the stream was closed OK:

Carry true.
Zero false.
A corrupt.

If the stream is not open:

Carry false.
Zero false.
A contains an error number (#0E).

If the close failed for any other reason:

Carry false.
Zero true.
A contains an error number.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

It is necessary to call this routine after using CAS OUT CHAR or CAS OUT DIRECT to ensure that all the data is written to the disc, to write the header to the start of the file and to give the file its true name.

If no data has been written to the file then it is abandoned and nothing is written to disc. This is for compatability with cassette routines.

When the file was opened it was given the type part of '.$$$'. This routine will rename the file to its true name and rename any existing version to have a '.BAK' type part. This ensures that any previous version of the file is automatically kept as a backup. Any existing '.BAK' version is deleted. If, when the file was opened, the caller did not specify the type part then AMSDOS will use the type part '.BAS' for BASIC files, '.BIN' for binary files and '.   ' for all other files, as specified by the file type field in the header.

If the actual length of the file is not a multiple of 128 bytes (a CP/M record) then a CP/M end of file character, #1A, is added to the file. This additional character is not recorded in the length of the file.

If writing is to be abandoned then CAS OUT OPEN should be called as this does not write any more data to disc.

The user may reclaim the buffer passed to CAS OUT OPEN after calling this routine.

Related entries:

CAS IN CLOSE (DISC)
CAS OUT ABANDON (DISC)
CAS OUT CLOSE
CAS OUT OPEN (DISC)

134: CAS OUT ABANDON #BC92

Close the output file immediately.

Action:

Abandon the output file and mark the write stream closed. Any unwritten data is discarded and not written to tape.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

This routine in intended for use after an error or in similar circumstances.

Related entries:

CAS IN ABANDON
CAS OUT ABANDON (DISC)
CAS OUT CLOSE
CAS OUT OPEN

134: CAS OUT ABANDON (DISC) #BC92

Close the output file immediately.

Action:

Abandon the output file and mark the write stream closed. Any unwritten data is discarded and not written to disc.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

This routine is intended for use after an error or similar circumstances.

If more than one 16K physical extent has already been written to disc then the file will appear in the disc directory with a type part of '.$$$'. Otherwise the file will disappear. This is because each 16K of a file requires a directory entry. A directory entry is not written to disc until the 16K has been written or a file is closed (CAS OUT CLOSE).

Related entries:

CAS IN ABANDON (DISC)
CAS OUT ABANDON
CAS OUT CLOSE (DISC)
CAS OUT OPEN (DISC)

135: CAS OUT CHAR #BC95

Write a character to the output file.

Action:

Add a character to the buffer for the write stream. If the buffer is already full then it is written to tape before the new character is inserted.

Entry conditions:

A contains the character to write.

Exit conditions:

If the character was written OK:

Carry true.
Zero false.
A corrupt.

If the file was not open as expected:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.

Always:

A, IX and other flags corrupt.
All other registers preserved.
Notes:

This routine can return two error numbers:

#00: The user hit escape.
#0E: The stream is not open for writing characters or the user hit escape previously.

If this routine returns the file not open as expected condition then either the user has hit escape previously or the file has been written using CAS OUT DIRECT. In either case, or if escape is pressed, the character sent will be discarded.

It is necessary to call CAS OUT CLOSE after sending all the characters to the file to ensure that the last block is written to the tape.

Once the routine has been called it is not possible to switch to directly writing the file.

Related entries:

CAS IN CHAR
CAS OUT CHAR (DISC) **CAS OUT DIRECT**
CAS OUT CLOSE **CAS OUT OPEN**

135: CAS OUT CHAR (DISC) #BC95

Write a character to an output file.

Action:

Add a character to the buffer for the write stream. If the buffer is already full then it is written to disc before the new character is inserted.

Entry conditions:

A contains the character to write.

Exit conditions:

If the character was written OK:

Carry true.
Zero false.
A corrupt.

If the stream is not open as expected:

Carry false.
Zero false.
A contains an error number (#0E).

If failed for any other reason:

Carry false.
Zero true.
A contains an error number.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

It is necessary to call CAS OUT CLOSE after sending all the characters to the file to ensure that the file is correctly written to disc.

Once the routine has been called it is not possible to switch to directly writing the file (CAS OUT DIRECT).

Related entries:

CAS IN CHAR (DISC)
CAS OUT CHAR
CAS OUT CLOSE (DISC)
CAS OUT DIRECT (DISC)
CAS OUT OPEN (DISC)

136: CAS OUT DIRECT #BC98

Write the output file directly from store.

Action:

Write the contents of store directly out to the output file.

Entry conditions:

HL contains the address of the data to write.

DE contains the length of the data to write.

BC contains the entry address (to go into the header).

A contains the file type (to go into the header).

Exit conditions:

If the file was written OK:

Carry true.
Zero false.
A corrupt.

If the file was not open as expected:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: a corrupt.

If the user hit escape:

Carry false.
Zero true.
In V1.1: A contains an error number (#00).
In V1.0: A corrupt.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

This routine can return two error numbers:

#00:	The user hit escape.
#0E:	The stream is not newly opened.

After writing the file it must be closed using CAS OUT CLOSE to ensure that the last block of the file is written to tape.

It is not possible to change the method for writing files from character output (using CAS OUT CHAR) to direct output (using CAS OUT DIRECT) or visa versa once the method has been chosen. Nor is it possible to directly write a file in two of more parts by calling CAS OUT DIRECT more than once - this will write corrupt data. Attempting to break these rules will result in a file not open as expected error.

Related entries:

CAS IN DIRECT **CAS OUT DIRECT (DISC)**
CAS OUT CLOSE **CAS OUT OPEN**

136: CAS OUT DIRECT (DISC) #BC98

Write the output file directly from store.

Action:

Write the contents of store directly out to the output file.

Entry conditions:

HL contains the address of the data to write (to go into the header).

DE contains the length of the data to write (to go into the header).

BC contains the entry address (to go into the header).

A contains the file type (to go into the header).

Exit conditions:

If the file was written OK:

Carry true.
Zero false.
A corrupt.

If the stream is not open as expected:

Carry false.
Zero false.
A contains an error number (#0E).

If failed for any other reason:

Carry false.
Zero true.
A contains an error number.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

After writing the file it must be closed using CAS OUT CLOSE to ensure that the file is written to disc.

It is not possible to change the method for writing files from character output (using CAS OUT CHAR) to direct output (using CAS OUT DIRECT) or visa versa once the method has been chosen. Nor is it possible to directly write a file in two of more parts by calling CAS OUT DIRECT more than once - this will write corrupt data.

Related entries:

CAS IN DIRECT (DISC)
CAS OUT CLOSE (DISC)
CAS OUT DIRECT
CAS OUT OPEN (DISC)

137: CAS CATALOG #BC9B

Generate a catalogue from tape.

Action:

Read file blocks to check their validity and print information about them on the screen.

Entry conditions:

DE contains the address of a 2K buffer to use.

Exit conditions:

If the cataloguing went OK:

Carry true.
Zero false.
A corrupt.

If the read stream was in use:

Carry false.
Zero false.
In V1.1: A contains an error number (#0E).
In V1.0: A corrupt.

Always:

BC, DE, HL, IX and other flags corrupt.
All registers preserved.

Notes:

This routine can only return one error number:

#0E: The stream is already in use.

This routine uses the read stream and so the stream must be closed when it is called. The read stream remains closed when this routine exits. The write stream is unaffected by this routine.

The prompt messages are turned on (see CAS NOISY) by this routine.

When cataloguing the Cassette Manager reads a header record, prints information from it and then reads the data record. This cycle repeats until the user hits the escape key. The information printed is as follows:

FILENAME block N T Ok

FILENAME is the name of the file on the tape, or 'Unnamed file' if the filename starts with a null (character #00).

N is the number of the block. Block 1 is normally the first block in a file.

T is a representation of the file type of the file. It is formed by adding #24 (the character '$') to the file type byte masked with #0F (to remove the version number field). The standard file types are thus:

$	a BASIC program file
%	a protected BASIC program file
*	an ASCII text file (default file type)
&	a binary file.
‘	a protected binary file

Other file types are possible but will not have been written by the BASIC in the on-board ROM. See section 8.4 for a description of the file type byte.

Ok is printed after the end of the data record. This shows that the data was read without errors and also serves to indicate the end of the data on tape (to help avoid over-recording a tape file).

Related entries:

CAS CATALOG (DISC)
CAS NOISY

137: CAS CATALOG (DISC) #BC9B

Display the disc directory

Action:

Display the disc directory for the current drive and current user. The directory is sorted into alphabetical order and displayed in as many columns as will fit in the current text window (stream #0). The size in Kbytes is shown along side each file. The total amount of free space on the disc is also shown.

Entry conditions:

DE contains the address of a 2K buffer to use.

Exit conditions:

If the cataloguing went OK:

Carry true.
Zero false.
A corrupt.

If failed for any reason:

Carry false.
Zero true.
A contains an error number.

Always:

BC, DE, HL, IX and other flags corrupt.
All registers preserved.

Notes:

Files marked SYS are not shown.

Files marked R/O are shown with a '*' after the filename.

Unlike the cassette version of this routine, the disc input stream is not required. (Note: BASIC abandons both the input and output streams when generating the catalogue.)

Related entries:

CAS CATALOG
|DIR

138: CAS WRITE #BC9E

Write a record to tape.

Action:

Write a record to the cassette. This routine is used by the higher level routines (CAS OUT CHAR, CAS OUT DIRECT and CAS OUT CLOSE) to write the header and data records that make up a tape file.

Entry conditions:

HL contains the address of the data to write.

DE contains the length of the data to write.

A contains the sync character to write at the end of the leader.

Exit conditions:

If the record was written OK:

Carry true.
A corrupt.

If an error occurred or the user hit escape:

Carry false.
A contains an error code.

Always:

BC, DE, HL, IX corrupt.
All other registers preserved.

Notes:

A data length of 0 passed to this routine is taken to mean 65536 bytes and all of the memory will be written to tape. (This is unlikely to be useful).

The data to be written may lie anywhere in RAM, even underneath a ROM.

The sync character is used to distinguish header records (sync is #2C) from data records (sync is #16). Other sync characters could be used but the resulting record would require special action to be taken to read it.

The error codes returned by this routine are:

0	Break	The user hit the escape key.
1	Overrun	The Cassette Manager was unable to get back to writing a bit fast enough.

Because reading and writing the tape requires stringent timing considerations interrupts are disabled whilst the tape is being written (potentially a period of over 5 minutes). It would be unpleasant to have the sound chip making a noise for all this time so the Sound Manager is shut down (SOUND RESET). When writing to the tape has finished interrupts are re-enabled.

The cassette motor is started by this routine (in case it is not already on) and restored to its previous state when writing is completed.

Related entries:

CAS CHECK
CAS READ

139: CAS READ #BCA1

Read a record from tape.

Action:

Read a whole record from the cassette. This routine is used by the higher level routines (CAS IN CHAR, CAS IN DIRECT and CAS CATALOG amongst others) to read the header and data records that make up a file.

Entry conditions:

HL contains the address to put the data read.

DE contains the length of the data to read.

A contains the sync character expected at the end of the leader.

Exit conditions:

If record was read OK:

Carry true.
A corrupt.

If an error occurred or the user hit escape:

Carry false.
A contains an error code.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

A data length of 0 passed to this routine is taken to mean 65536 bytes. (This is not useful).

It is not necessary to read a whole record from tape. If the length passed is less than the actual length of the record then only the number of bytes will be read. Trying to read more bytes from a record than were written will produce an error, usually an overflow error (see below).

The sync character is used to distinguish header records (sync is #2C) from data records (sync is #16). Other sync characters could be used if the record was written that way.

The error codes returned by this routine are:

0	Break	The user hit the escape key.
1	Overrun	The Cassette Manager found a bit that was too long to read.
2	CRC	A CRC failure was detected.

The cassette motor is started by this routine (in case it is not already on) and restored to its previous state when reading is completed.

Because reading the tape requires stringent timing constraints, interrupts are disabled whilst the tape is being read (potentially a period of over 5 minutes). It would be unpleasant to have the sound chip making a noise for all this time so the Sound Manager is shut down (SOUND RESET). When reading from the tape has finished interrupts are re-enabled.

Related entries:

CAS CHECK
CAS WRITE

140: CAS CHECK #BCA4

Compare a record on tape with the contents of store.

Action:

Check that a tape record contains a correct version of the data supplied. This routine is intended to be used after writing records to check that they were written correctly.

Entry conditions:

HL contains the address of the data to check.

DE contains the length of the data to check.

A contains the sync character expected at the end of the leader.

Exit conditions:

If the record checked OK:

Carry true.
A corrupt.

If an error occurred or the user hit escape:

Carry false.
A contains an error code.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

A data length of 0 passed to this routine is taken to mean 65536 bytes. (This is bound to produce a check failure).

It is not necessary to check the whole of a record on tape. If the length passed is less than the actual length of the record then only the number of bytes will be checked. Trying to check more bytes in a record than were written will produce an error of some sort (see below).

The data to be checked may lie anywhere in RAM, even underneath a ROM.

The sync character is used to distinguish header records (sync is #2C) from data records (sync is #16). Other sync characters could be used.

The error codes returned by this routine are:

0	Break	The user hit the escape key.
1	Overrun	The Cassette Manager found a bit that was too long to read.
2	CRC	A CRC failure was detected.
3	Different	The data read from tape did not agree with that in memory.

The cassette motor is started by this routine (in case it is not already on) and restored to its previous state when checking is completed.

Because reading the tape requires stringent timing constraints, interrupts are disabled whilst the tape is being checked (potentially a period of over 5 minutes). It would be unpleasant to have the sound chip making a noise for all this time so the Sound Manager is shut down (SOUND RESET). When checking has finished interrupts are re-enabled.

Related entries:

CAS READ
CAS WRITE

141: SOUND RESET #BCA7

Reset the Sound Manager.

Action:

Re-initialize the Sound Manager - shut the sound chip up and clear all queues.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The sound queues are cleared. Any current sound is stopped. The sound generator chip is silenced.

This routine enables interrupts.

Related entries:

SOUND HOLD

142: SOUND QUEUE #BCAA

Add a sound to a sound queue.

Action:

Try to add a sound to the sound queue of one or more channels. If the sound queue of any of the channels is full then no sound will be issued to any channel.

Entry conditions:

HL contains the address of a sound program which must lie in the central 32K of RAM.

Exit conditions:

If the sound was added to the queue(s):

Carry true.
HL corrupt.

If at least one queue was full:

Carry false.
HL preserved.

Always:

A, BC, DE, IX and other flags corrupt.
All other registers preserved.

Notes:

The sound program is laid out as follows:

Byte 0:	Channels to use and rendezvous requirements.
Byte 1:	Amplitude envelope to use.
Byte 2:	Tone envelope to use.
Bytes 3..4:	Tone period.
Byte 5:	Noise period.
Byte 6:	Initial amplitude.
Bytes 7..8:	Duration or envelope repeat count.

All values in the sound program are masked into the appropriate range before being used.

The channels to issue the sound on are encoded into byte 0 as follows:

Bit 0:	Issue on channel A.
Bit 1:	Issue on channel B.
Bit 2:	Issue on channel C.

The rendezvous requirements are encoded into byte 0 as follows:

Bit 3:	Rendezvous with channel A.
Bit 4:	Rendezvous with channel B.
Bit 5:	Rendezvous with channel C.
Bit 6:	Hold until released.
Bit 7:	Flush queue.

A channel will ignore an order to rendezvous with itself. Sounds issued on multiple channels implicitly rendezvous with each other. Sounds that are ordered to rendezvous will be issued to the sound generator starting at the same time.

Setting the hold bit prevents the sound from running until it is released by calling SOUND RELEASE (or a routine having a similar effect). Setting the flush bit will empty the queue and abandon any currently active sound thus allowing the new sound to start immediately.

The amplitude envelope is in the range 0..15. Envelopes 1..15 are the amplitude envelopes that can be set using SOUND AMPL ENVELOPE. Envelope 0 means use no amplitude envelope, simply hold the initial amplitude for 2 seconds or the duration specified.

The tone envelope is in the range 0..15. Envelopes 1..15 are the tone envelopes that can be set using SOUND TONE ENVELOPE. Envelope 0 means use no tone envelope, simply hold the initial tone.

A tone period of 0 means do not generate any tone. Tone periods in the range 1..4095 specify the period of the tone in 8 microsecond units.

The noise period is in the range 0..31. Noise periods 1..31 specify the period of the noise component of a sound. A noise period of 0 means use no noise.

The initial amplitude is in the range 0..15. Amplitude 0 being no initial sound, amplitude 15 being the maximum volume.

Bytes 7 and 8 store the sound time. If this is zero then the amplitude envelope is obeyed once. If the sound time is negative then the amplitude envelope is obeyed minus the sound time number of times (i.e. 1..32768 times). If the sound time is positive but not zero then it is taken to be the duration of the sound in 1/100s of a second.

If a duration is specified when an amplitude envelope is in use then the duration given sets the length of the sound. If the duration is longer than the envelope then the final amplitude of the envelope is sustained until the duration expires. Tone envelopes are treated in much the same way as amplitude envelopes except that they never specify the length of the sound.

The sound event that is run when a sound queue has a free slot is disarmed on the channels specified in this command.

All sounds currently held by SOUND HOLD are automatically released when this routine is called. Also, the sound queue event is disarmed (see SOUND ARM EVENT). SOUND QUEUE may enable interrupts.

Related entries:

SOUND ARM EVENT
SOUND CHECK
SOUND RELEASE

143: SOUND CHECK #BCAD

Ask if there is space in a sound queue.

Action:

Ask the state of a sound channel. The status includes the number of free spaces in the sound queue and whether the channel is held.

Entry conditions:

A contains the bit for the channel to test.

Exit conditions:

A contains the channel status.

BC, DE, HL and flags corrupt.
All other registers preserved.

Notes:

The channel to ask the status of is encoded as follows:

Bit 0:	Ask about channel A.
Bit 1:	Ask about channel B.
Bit 2:	Ask about channel C.

If more than one bit is set then the status of only one channel is returned. The channels are tested in the order given above.

The status returned is encoded as follows:

Bits 0..2:	Contain the number of free slots in the channel's sound queue.
Bit 3:	The channel is awaiting a rendezvous with channel A.
Bit 4:	The channel is awaiting a rendezvous with channel B.
Bit 5:	The channel is awaiting a rendezvous with channel C.
Bit 6:	The channel is held.
Bit 7:	The channel is active (producing a sound).

Calling this routine disarms the sound queue event that occurs when the queue has a free slot for the channel returned (see SOUND ARM EVENT).

This routine may enable interrupts.

Related entries:

SOUND ARM EVENT
SOUND QUEUE

144: SOUND ARM EVENT #BCB0

Set up an event to be run when a sound queue becomes empty.

Action:

Arm the sound event to be run when a free slot occurs in a channel's sound queue.

Entry conditions:

A contains the bit for the channel to arm.
HL contains the address of an event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The channel for which to arm the event is encoded as follows:

Bit 0:	Arm channel A.
Bit 1:	Arm channel B.
Bit 2:	Arm channel C.

If more than one bit is set then only one channel is armed. The channels are armed in the order given above.

The event block passed must be initialized (by KL INIT EVENT).

The event will be 'kicked' when a free slot occurs in the queue. If there is a free slot in the queue when this routine is called then the event will be 'kicked' immediately.

The sound event is disarmed automatically when SOUND QUEUE or SOUND CHECK is called. It is also disarmed when the event is run. Thus, the event routine will need to rearm the sound vent to keep it running continuously.

This routine may enable interrupts.

Related entries:

KL INIT EVENT
SOUND CHECK
SOUND QUEUE

145: SOUND RELEASE #BCB3

Allow sounds which are individually held to start.

Action:

Release held sounds on a number of channels. This allows sounds that were marked with a hold bit when they were set up by SOUND QUEUE to start (other factors willing).

Entry conditions:

A contains bits for the channels to release.

Exit conditions:

AF, BC, DE, HL and IX corrupt.

All other registers preserved.

Notes:

The channels to release are encoded as follows:

Bit 0:	Release channel A.
Bit 1:	Release channel B.
Bit 2:	Release channel C.

All channels that are specified are released.

All sounds currently held by SOUND HOLD are automatically released.

This routine may enable interrupts.

Related entries:

SOUND QUEUE

146: SOUND HOLD #BCB6

Stop all sounds in midflight.

Action:

This stops all sounds immediately. The sounds can be started again by calling SOUND CONTINUE.

Entry conditions:

No conditions.

Exit conditions:

If a sound was active:
    Carry true.

If no sound was active:
    Carry false.

Always:
    A, BC, HL and other flags corrupt.
    All other registers preserved.

Notes:

Sounds that are held by this routine are automatically restarted when SOUND QUEUE or SOUND RELEASE are called as well as when SOUND CONTINUE itself is called.

The sound is stopped by halting the execution of sound and tone envelopes and setting the sound chip volume to zero for all channels. When the sound is restarted it will continue from as near where it was stopped as is possible.

This routine enables interrupts.

Related entries:

SOUND CONTINUE
SOUND RESET

147: SOUND CONTINUE #BCB9

Restart sounds after they have all been held.

Action:

Allows sounds that have been held by calling SOUND HOLD to continue.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and IX corrupt.
All other registers preserved.

Notes:

If no sounds are held then no action is taken.

This routine may enable interrupts.

Related entries:

SOUND HOLD
SOUND RELEASE

148: SOUND AMPL ENVELOPE #BCBC

Set up an amplitude envelope.

Action:

Set up one of the 15 user programmable amplitude (volume) envelopes.

Entry conditions:

A contains an envelope number.

HL contains the address of an amplitude data block.

Exit conditions:

If envelope has been set OK:

Carry true.
HL contains the address of the data block + 16.
A and BC corrupt.

If the envelope number is invalid:

Carry false.
A, B and HL preserved.

Always:

DE and other flags corrupt.
All other registers preserved.

Notes:

The envelope to set up is specified by a number in the range 1..15. No envelope is set up if a number outside this range is passed.

The amplitude data block is copied into the amplitude envelope. The data block may lie in ROM or in RAM. It may not lie in RAM hidden underneath a ROM.

The amplitude data block has the following layout:

Byte 0:	Count of sections in the envelope.
Bytes 1..3:	First section of the envelope.
Bytes 4..6:	Second section of the envelope.
Bytes 7..9:	Third section of the envelope.
Bytes 10..12:	Fourth section of the envelope.
Bytes 13..15:	Fifth section of the envelope.

The first byte of the amplitude data block specifies the number of sections used in the envelope. Sections not used need not be set up. An envelope using no sections has a special meaning - hold a constant volume lasting for 2 seconds.

The number of sections to use is not checked, if a number outside the range 0..5 is supplied then this will have unpredictable effects. This should be avoided.

Each section of the amplitude data block can specify either a hardware or a software envelope. This is indicated by the first byte of the section.

A software envelope section is laid out as follows:

Byte 0:	Step count.
Byte 1:	Step size.
Byte 2:	Pause time.

The fact that this is a software envelope section rather than a hardware envelope section is indicated by byte 0 not having bit 7 set.

If the step count is in the range 1..27 then the step size is added to the volume that number of times with a wait equal to the pause time in 1/100s of a second after each addition.

If the step count is 0 the step size is taken to be an absolute volume setting. A single wait of the pause time in 1/100s of a second is made.

After calculating the new volume this is masked with #0F to make sure it is legal. Thus, all arithmetic on the volume is carried out modulo 16.

A pause time of 0 is taken to mean 256 1/100s of a second.

A hardware envelope section is laid out as follows:

Byte 0:	Envelope shape.
Byte 1..2:	Envelope period.

The fact that this is a hardware envelope section rather than a software envelope section is indicated by byte 0 having bit 7 set.

The envelope shape (masked with #7F) is sent to register 13 of the sound generator. This sets the shape of the hardware envelope and whether it repeats (see Appendix IX for details).

The envelope period is sent to registers 11 and 12 of the sound generator. These set the length of the hardware envelope (see Appendix IX for details).

The section after a hardware section should be a pause long enough to allow the hardware envelope to operate. A pause can be constructed using a software envelope with a step size of 0 and with the repeat count and pause time juggled to give the right total time.

There is no protection against changing an envelope whilst it is in use. This could have unpredictable effects and should be avoided.

The length of the sound can either be determined by the duration supplied when the sound is queued or by the envelope terminating (see SOUND QUEUE). If a duration is set that is shorter than the envelope then the envelope is truncated. If the duration is longer than the envelope then the final volume is sustained until the duration expires.

Related entries:

SOUND A ADDRESS
SOUND TONE ENVELOPE

149: SOUND TONE ENVELOPE #BCBF

Set up a tone envelope.

Action:

Set up one of the 15 user programmable tone envelopes.

Entry conditions:

A contains an envelope number.
HL contains the address of a tone data block.

Exit conditions:

If the envelope has been set up OK:

Carry true.
HL contains the address of the data block + 16.
A and BC corrupt.

If the envelope number is invalid:

Carry false.
A, BC and HL preserved.

Always:

DE and other flags corrupt.
All other registers preserved.

Notes:

The envelope to set up is specified by a number in the range 1..15. No envelope is set up if a number outside this range is passed.

The tone data block is copied into the tone envelope. The data block may lie in ROM or in RAM. It may not lie in RAM underneath a ROM.

The tone data block has the following layout:

Byte 0:	Count of sections in the envelope.
Bytes 1..3:	First section of the envelope.
Bytes 4..6:	Second section of the envelope.
Bytes 7..9:	Third section of the envelope.
Bytes 10..12:	Fourth section of the envelope.
Bytes 13..15:	Fifth section of the envelope.

The first byte of the tone data block (masked with #7F) specifies the number of sections used in the envelope. Sections not used need not be set up. An envelope using no sections will not alter the tone (i.e. no enveloping). The number of sections to use is not checked, if a number outside the range 0..5 is supplied then this will have unpredictable effects. This should be avoided.

The top bit, bit 7, of the count is used to indicate a repeating envelope. If this bit is set then when the last section of the envelope finishes the first will be used again.

Each section of the tone data block is laid out as follows:

Byte 0:	Step count.
Byte 1:	Step size.
Byte 2:	Pause time.

If the step count lies in the range #00..#EF then the section is a relative section. The step size is sign extended (bit 7 is copied to bits 8..15) and is added to the current tone period the number of times specified by the step count. After each addition a wait of the pause time in 1/100s of a second is made. The sound chip only uses the lower 12 bits of the tone period so all arithmetic is carried out modulo #1000.

A step count of 0 is taken to mean 1 step whilst a pause time of 0 is taken to mean 256 1/100s of a second.

If the step count lies in the range #F0..#FF then the section is an absolute section. The least significant four bits of the step count are taken to be the most significant byte of the tone period and the step size is taken to be the least significant byte. This tone period is set immediately and is followed by a pause whose length is set by the pause time in 1/100s of a second.

There is no protection against changing an envelope whilst it is in use. This could have unpredictable effects and should be avoided.

If the tone envelope finishes before the end of the sound (as set when the sound was queued) then the final tone is held. i.e. The tone envelope does not affect the length of the sound.

Related entries:

SOUND AMPL ENVELOPE
SOUND T ADDRESS

150: SOUND A ADDRESS #BCC2

Get the address of an amplitude envelope.

Action:

Ask where the data area for an amplitude envelope is stored.

Entry conditions:

A contains an envelope number.

Exit conditions:

If the envelope was found OK:

Carry true.
HL contains the address of the amplitude envelope.
BC contains the length of an envelope (16 bytes).

If the envelope number was invalid:

Carry false.
HL corrupt.
BC preserved.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

The envelope number must lie in the range 1..15.

The amplitude envelope is laid out as described in SOUND AMPL ENVELOPE.

Related entries:

SOUND AMPL ENVELOPE
SOUND T ADDRESS

151: SOUND T ADDRESS #BCC5

Get the address of a tone envelope.

Action:

Ask where the data area for the tone envelope is stored.

Entry conditions:

A contains an envelope number.

Exit conditions:

If the envelope was found OK:

Carry true.
HL contains the address of the tone envelope.
BC contains the length of an envelope (16 bytes).

If the envelope number was invalid:

Carry false.
HL corrupt.
BC preserved.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

The envelope number must lie in the range 1..15.

The tone envelope is laid out as described in SOUND TONE ENVELOPE.

Related entries:

SOUND A ADDRESS
SOUND TONE ENVELOPE

152: KL CHOKE OFF #BCC8

Reset the Kernel - clears all event queues etc.

Action:

This entry completely clears all event queues, the various timer and frame flyback lists and so on. The effect is to dispose of any pending synchronous events and to halt all timer related functions other than sound generation and keyboard scanning.

Entry conditions:

No conditions.

Exit conditions:

B contains the ROM select address of the current foreground ROM (if any).

DE contains the address at which the current foreground ROM was entered.

C contains the ROM select address for a RAM foreground program.

AF and HL corrupt.
All other registers preserved.

Notes:

If the current foreground program is in RAM then the ROM select address and entry point returned are both zero. i.e. The default ROM (ROM 0) at its entry address.

KL CHOKE OFF forms part of the close down required before a new RAM foreground program is loaded, as is required by MC BOOT PROGRAM.

The close down must ensure that there are no interrupt or other events active and using memory which might be damaged by loading a new program into memory. In the complete close down MC BOOT PROGRAM does:

SOUND RESET	to kill off sound generation
an OUT to I/O port #F8FF	to reset any external interrupt sources.
KL CHOKE OFF	to kill off events etc.
KM RESET	to reset any keyboard indirections and the break event.
TXT RESET	to reset any Text VDU indirections.
SCR RESET	to reset any screen indirections.

The values returned by KL CHOKE OFF are used by MC BOOT PROGRAM if the program load fails.

This information is included for the reader's interest. MC BOOT PROGRAM is the recommended means of loading and entering a RAM foreground program. MC START PROGRAM is the recommended means of entering a ROM foreground program, or a RAM foreground program which has already been loaded.

KL CHOKE OFF disables interrupts.

Related entries:

MC BOOT PROGRAM
MC START PROGRAM

153: KL ROM WALK #BCCB

Find and initialize all background ROMs.

Action:

Background ROMs provide support for expansion hardware or augment the software facilities of the machine. If the facilities provided by the background ROMs are to be available, the foreground program must initialize them. This routine finds and initializes all background ROMs.

Entry conditions:

DE contains address of the first usable byte of memory (lowest address).

HL contains address of the last usable bytes of memory (highest address).

Exit conditions:

DE contains the address of the new first usable byte of memory.

HL contains the address of the new last usable byte of memory.

AF and BC corrupt.

All other registers preserved.

Notes:

When a foreground ROM program is entered it is passed the addresses of the first and last bytes in memory which it may use. The area of memory outside this is used to store firmware variables, the stack, the jumpblocks and the screen memory. From the area available for the foreground program to use, the areas for background programs to use must be allocated.

The foreground program should initialize background ROMs at an early stage, before it uses the memory it is given. It may choose whether to enable background ROMs or not. KL INIT BACK may be used to initialize a particular background ROM or this routine may be used to initialize all available background ROMs.

KL ROM WALK inspects the ROMs at ROM select addresses in the range 1..7 in V1.0 firmware and 0..15 in V1.1 firmware. The power-up initialization entry of each background ROM found is called (unless it is the current foreground ROM in V1.1 firmware). This entry may allocate some memory for the background ROM's use by adjusting the values in DE and HL before returning. Once the ROM has been initialized the Kernel adds it to the list of external command servers, and notes the base of the area which the ROM has allocated at the top of memory (if any). Subsequent FAR CALLs to entries in the ROM will automatically set the IY index register to point at the ROMs upper memory area.

See section 10.4 for a full description of background ROMs.

Related entries:

KL FIND COMMAND
KL INIT BACK
KL LOG EXT

154: KL INIT BACK #BCCE

Initialize a particular background ROM.

Action:

Background ROMs provide support for expansion hardware or augment the software facilities of the machine. If the facilities provided by the background ROMs are to be available the foreground program must initialize them. This routine selects and initializes a particular background ROM.

Entry conditions:

C contains the ROM select address of the ROM to initialize.

DE contains address of the first usable byte of memory (lowest address).

HL contains address of the last usable byte of memory (highest address).

Exit conditions:

DE contains the address of the new first usable byte of memory.

HL contains the address of the new last usable byte of memory.

AF and B corrupt.

All other registers preserved.

Notes:

The ROM select address must be in the range 1..7 in V1.0 and 0..15 in V1.1 firmware and the ROM at this address must be a background ROM or the request will be ignored. In V1.1 firmware the request will be ignored if the ROM is the current foreground ROM.

When the foreground program is entered it is passed the addresses of the first and last bytes in memory which it may use. The area of memory outside this is used to store firmware variables, the stack, the jumpblocks and the screen memory. From the area available for the foreground program to use, the areas for background programs to use must be allocated.

The foreground program should initialize background ROMs at an early stage, before it uses the memory it is given. It may choose whether to enable background ROMs or not. KL ROM WALK may be used to initialize all available ROMs or this routine may be used to initialize particular ROMs.

This routine causes the background ROM's power-up initialization entry to be called. This entry may allocate some memory for the background ROM's use by adjusting the values in DE and HL before returning. Once the ROM has been initialized the Kernel adds it to the list of external command servers, and notes the base of the area which the ROM has allocated to itself at the top of memory (if any). Subsequent FAR CALLs to entries in the ROM will automatically set the IY index register to point at the ROM's upper memory area.

See section 10.4 for a full description of background ROMs.

Related entries:

KL FIND COMMAND
KL LOG EXT
KL ROM WALK

155: KL LOG EXT #BCD1

Introduce an RSX to the firmware.

Action:

RSXs (Resident System Extensions) are similar to background ROMs, but are loaded into RAM. This routine must be called to include the RSX on the Kernel's list of external command servers.

Entry conditions:

BC contains the address of the RSX's command table.

HL contains the address of a 4 byte area of RAM for the Kernel's use.

Exit conditions:

DE corrupt.

All other registers preserved.

Notes:

Both the RSX's command table and the Kernel's storage area must lie in the central 32K of memory, i.e. not under a ROM.

The format of command table is described in section 10.2 and RSXs are discussed in section 10.5.

Related entries:

KL FIND COMMAND
KL INIT BACK

156: KL FIND COMMAND #BCD4

Search for an RSX, background ROM or foreground ROM to process a command.

Action:

All expansion ROMs and RSXs have command tables of the same form. This routine searches all RSXs and background ROMs on the Kernel's list of external command servers looking for a match for the given command name. If the name is found, then the 'far address' of the associated routine is returned. If the command is not a background or RSX command then all the foreground ROMs that can be found are searched for a foreground program with the given name. If a foreground program is found then the system immediately enters it.

Entry conditions:

HL contains the address of the command name to search for.

Exit conditions:

If an RSX or background ROM command was found:

Carry true.
C contains the ROM select address.
HL contains the address of the routine.

If the command was not found:

Carry false.
C and HL corrupt.

Always:

A, B and DE corrupt.
All other registers preserved.

Notes:

The command name passed must be in RAM but may lie underneath a ROM. The name may be any number of characters long but only the first 16 characters are significant. All alphabetic characters in the name should be in upper case and the last character of the name should have bit 7 set.

The ROM select and routine addresses returned are suitable for calling KL FAR PCHL.

The list of external command servers is generated as background ROMs and RSXs are initialized (see KL ROM WALK, KL INIT BACK and KL LOG EXT). The command tables are scanned in the opposite order to that in which the command servers were introduced. Thus, RSXs will tend to take precedence over background ROMs, since RSX's are, in general, initialized after background ROMs. Background ROMs are normally initialized in reverse order of ROM select addresses, so lower numbered ROMS will take precedence over higher.

See section 10.2 for a description of the format of expansion ROM command tables.

The first entry in a background ROM's command name table (the one associated with the power-up entry) may be used as the ROM's name. KL FIND COMMAND may be used, therefore, to find out whether a particular background ROM has been initialized.

When searching for a foreground program, ROMs are inspected starting with ROM 0 and working upwards. The search ceases when the first unused ROM address greater than 0 on V1.0 firmware and greater than 15 on V1.1 firmware is found.

The on-board BASIC may be entered by searching for and invoking the command 'BASIC'.

If a foreground ROM command is found the ROM is entered unconditionally this routine never returns.

Related entries:

KL INIT BACK
KL LOG EXT
KL ROM WALK
MC START PROGRAM

157: KL NEW FRAME FLY #BCD7

Initialize and put a block onto the frame flyback list.

Action:

The Kernel maintains a list of events to be kicked each time frame flyback occurs. The routine initializes a block and adds it to the list.

Entry conditions:

HL contains the address of the frame flyback block.

B contains the event class.

C contains the ROM select address of the event routine.

DE contains the address of the event routine.

Exit conditions:

AF, DE and HL corrupt.

All other registers preserved.

Notes:

The frame flyback block is 9 bytes long and must lie in the central 32K of RAM. The last 7 bytes of the frame flyback block are an event block which is initialized to reflect the parameters passed in B, C and DE (see KL INIT EVENT). The exact layout of a frame flyback block is described in Appendix X.

The frame flyback block is appended to the frame flyback list if it is not already on it.

This routine enables interrupts.

Related entries:

KL ADD FRAME FLY
KL DEL FRAME FLY
KL INIT EVENT

158: KL ADD FRAME FLY #BCDA

Put a block onto the frame flyback list.

Action:

The Kernel maintains a list of events to be kicked each time frame flyback occurs. This routine adds a block to the list.

Entry conditions:

HL contains the address of the frame flyback block.

Exit conditions:

AF, DE and HL corrupt.

All other registers preserved.

Notes:

The frame flyback block is 9 bytes long and it must lie in the central 32K of RAM. The last 7 bytes of the frame flyback block are an event block which must be initialized separately before calling this routine. The exact layout of a frame flyback block is described in Appendix X.

The block is appended to the frame flyback list if it is not already on it.

The routine enables interrupts.

Related entries:

KL DEL FRAME FLY
KL INIT EVENT
KL NEW FRAME FLY

159: KL DEL FRAME FLY #BCDD

Remove a block from the frame flyback list.

Action:

The Kernel maintains a list of events to be kicked each time frame flyback occurs. This routine removes a block from the list.

Entry conditions:

HL contains the address of the frame flyback block.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

This routine does nothing if the block is not on the list.

Removing a block from the list only prevents the event being kicked again. It does not affect any outstanding frame flyback events.

This routine enables interrupts.

Related entries:

KL ADD FRAME FLY
KL NEW FRAME FLY

160: KL NEW FAST TICKER #BCE0

Initialize and put a block onto the fast ticker list.

Action:

The Kernel maintains a list of events to be kicked each time the 1/300th of a second timer interrupt occurs. This is known as the fast ticker list. This routine initializes a block and adds it to the list.

Entry conditions:

HL contains the address of the fast ticker block.

B contains the event class.

C contains the ROM select address of the event routine.

DE contains the address of the event routine.

Exit conditions:

AF, DE and HL corrupt.

All other registers preserved.

Notes:

The fast ticker block is 9 bytes long and must lie in the central 32K of RAM. The last 7 bytes of the fast ticker block are an event block which must be initialized to reflect the parameters passed in B, C and DE (see KL INIT EVENT). The exact layout of a fast ticker block is described in Appendix X.

The fast ticker block is appended to the fast ticker list if it is not already on it.

The fast ticker facility is not intended for general use. However, it does allow relatively short times to be measured giving greater resolution than the general ticker facilities.

This routine enables interrupts.

Related entries:

KL ADD FAST TICKER
KL ADD TICKER
KL DEL FAST TICKER
KL INIT EVENT
KL TIME PLEASE

161: KL ADD FAST TICKER #BCE3

Put a block onto the fast ticker list.

Action:

The Kernel maintains a list of events to be kicked each time the 1/300th of a second timer interrupt occurs. This is known as the fast ticker list. This routine adds a block to the list.

Entry conditions:

HL contains the address of the fast ticker block.

Exit conditions:

AF, DE and HL corrupt.

All other registers preserved.

Notes:

The fast ticker block is 9 bytes long and must lie in the central 32K of RAM. The last 7 bytes of the fast ticker block are an event block which must be initialized before calling this routine. The exact layout of a fast ticker block is described in Appendix X.

The fast ticker block is appended to the fast ticker list if it is not already on it.

The fast ticker facility is not intended for general use. However, it does allow relatively short times to be measured giving greater resolution than the general ticker facilities.

This routine enables interrupts.

Related entries:

KL ADD TICKER
KL DEL FAST TICKER
KL INIT EVENT
KL NEW FAST TICKER
KL TIME PLEASE

162: KL DEL FAST TICKER #BCE6

Remove a block from the fast ticker list.

Action:

The Kernel maintains a list of events to be kicked each time the 1/300th of a second timer interrupt occurs. This is known as the fast ticker list. This routine removes a block from the list.

Entry conditions:

HL contains the address of the fast ticker block.

Exit conditions:

AF, DE and HL corrupt.

All other registers preserved.

Notes:

This routine does nothing if the block is not on the list.

Removing a block from the list only prevents the event from being kicked again. It does not affect any outstanding fast ticker events.

This routine enables interrupts.

Related entries:

KL ADD FAST TICKER
KL DEL TICKER
KL NEW FAST TICKER

163: KL ADD TICKER #BCE9

Put a block onto the ticker list.

Action:

The general purpose timing facility measures time in 1/50th of a second units. The Kernel maintains a list of tick blocks each of which contains a count and a recharge value. Every 1/50th of a second the Kernel processes all the tick blocks, decrementing the count entry of each. If the count entry of a block becomes zero the event contained in the block is 'kicked', and the count is set to the recharge value.

Entry conditions:

HL contains the address of the tick block.

DE contains the initial value for the count entry.

BC contains the value of the recharge entry.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The tick block is 13 bytes long and must lie in the central 32K of memory. The last 7 bytes of the tick block are an event block which must be initialized before this routine is called. The exact layout of a tick block is described in Appendix X.

The count and recharge entries in the block are set. The block is then appended to the tick list if it is not already on the list. This routine may be used, therefore, to change the count and recharge entries of an existing block.

Blocks with a count entry of zero are ignored when the list is processed. Setting a recharge value of zero, therefore, sets up the block as a 'one shot timer'. Since it takes the Kernel time to ignore a tick block, any redundant blocks should be removed from the list as soon as possible.

It is not possible to predict, particularly with synchronous events, how long it will be after the 'kick' before the event routine is actually called. Notwithstanding these delays, the ticker may be used to obtain an exact number of 'kicks' in a given period since the recharge mechanism immediately resets the count. The event counting mechanism will ensure that 'kicks' are not missed, provided that there are never more than 127 outstanding at once.

This routine enables interrupts.

Related entries:

KL ADD FAST TICKER
KL DEL TICKER
KL INIT EVENT

164: KL DEL TICKER #BCEC

Remove block from the tick list.

Action:

If a given block is on the tick list it is removed. The contents of the block are not affected.

Entry conditions:

HL contains the address of the tick block.

Exit conditions:

If the tick block was found on the tick list:

Carry true.
DE contains the count remaining before the next event.

If the tick block was not found on the tick list:

Carry false.
DE corrupt.

Always:

A, HL and other flags corrupt.
All other registers preserved.

Notes:

The contents of the block are not affected by removing it from the list. In particular the continued processing of outstanding events is not affected. The block could be put back on the list at a later date and it could continue counting where it left off.

This routine enables interrupts.

Related entries:

KL ADD TICKER
KL DEL FAST TICKER

165: KL INIT EVENT #BCEF

Initialize an event block.

Action:

Initialize all entries in an event block.

Entry conditions:

HL contains the address of the event block.

B contains the event class.

C contains the ROM select address of the event routine.

DE contains the address of the event routine.

Exit conditions:

HL contains the address of the event block + 7.

All other registers preserved.

Notes:

The event block is 7 bytes long and must lie in the central 32K of RAM. The layout of an event block is described in Appendix X. See section 12 for a general discussion of events.

The ROM select and address of the routine are the 'far address' of the event routine (see section 2).

The event class is bit significant as follows:

Bit 0:	Near address.
Bits 1..4:	Synchronous event priority.
Bit 5:	Must be zero.
Bit 6:	Express event.
Bit 7:	Asynchronous event.

If the asynchronous event bit is set then the event is an asynchronous event, otherwise it is a synchronous event. Asynchronous events do not have priorities and so the priority field is ignored.

If the express event bit is set then the event is an express event. The meaning of this depends on whether the event is synchronous or asynchronous.

All express synchronous events have higher priorities than any normal synchronous event. The priority of a synchronous event is encoded in bits 1..4 of the class, the higher the number the greater the priority. No event may have priority 0. The processing of normal synchronous events may be disabled (by calling KL EVENT DISABLE), while that if express synchronous events may not.

An express asynchronous event will have its event routine called directly from the interrupt path. A normal asynchronous event is processed just before returning from the interrupt (with interrupts enabled).

If the near address bit is set then the event routine is located either in the lower ROM or in the central 32K of RAM. The ROM select address is ignored and the routine is called directly, rather than through the FAR CALL mechanism, thus reducing the event processing overhead. Where possible, asynchronous events should be at 'near addresses'. Express asynchronous events must always be at 'near addresses'.

Event blocks appear in various other blocks handled by the Kernel, including frame flyback, fast ticker and tick blocks. This routine is used to initialize the event block parts of these.

The bytes after the last byte of the event block, even where the block forms part of another block, are not used by the Kernel. When the event routine is called the address of the block is passed to it, so the user may append further information about the event to the block. This allows several similar events to share the same event routine, each event having its 'own' variables appended to its event block.

The event routine has the following entry and Exit conditions:

Entry:

If the event routine is at a 'far address':

HL contains the address of byte 5 of the event block
(so any appended data can start at address HL + 2).

If the event routine is at a 'near address':

DE contains the address of byte 6 of the event block
(so any appended data can start at address DE + 1).

Exit:

AF, BC, DE and HL corrupt.
All other registers preserved.

The event routine may use the IX and IY registers but must preserve them. It may not use the second register set. Express asynchronous events may not enable interrupts.

KL INIT EVENT enables interrupts.

Related entries:

KL DEL SYNCHRONOUS
KL DISARM EVENT
KL EVENT
KL NEW FAST TICKER
KL NEW FRAME
FLY KL NEW TICKER
KL SYNC RESET

166: KL EVENT #BCF2

'Kick' an event block.

Action:

The event mechanism arranges that an event routine be called in response to each 'kick' of an event block. KL EVENT performs the 'kick'.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

Unlike the vast majority of Kernel routines this routine may be called from the interrupt path. Because the LOW JUMP instruction in the main firmware jumpblock enables interrupts the user may pick the address part of the 'low address' out of the jumpblock and mask off the top two bits to extract the address in the lower ROM of KL EVENT. The following code does this:

```
LD DE,(KL_EVENT+1)            ;extract address of LOW JUMP
RES 7,D                       ;remove upper ROM state from 'low address'
RES 6,D                       ;remove lower ROM state from 'low address'
CALL PCDE_INSTRUCTION         ;CALL KL EVENT
```

(If the user is going to perform this operation repeatedly it is recommended that the address should be extracted once and should be stored somewhere).

The effect of the 'kick' depends on the event count in the event block:

Count < 0: The event is disarmed, and kicking it has no effect.

Count > 0: There are other kicks outstanding and the event is being processed. This kick simply increments the count (unless it has already reached the maximum of 127). Once event processing has begun it continues until the count becomes zero or the event is disarmed.

Count = 0: The event is armed but event processing is not active.

The count is incremented and event processing initiated.

How event processing is initiated depends on the event class.

Synchronous Events.

Synchronous events are added to the synchronous event queue in priority order. It is the responsibility of the foreground program to process the synchronous event queue regularly.

Synchronous event routine are called when the foreground program calls KL DO SYNC, the event count is then dealt with when KL DONE SYNC is called.

Asynchronous Events.

a. Not in the Interrupt Path

The event routine is called immediately. When the routine returns, if the event count is greater than zero it is decremented. If the count is still greater than zero then the procedure is repeated.

b. In the Interrupt Path - Normal Asynchronous Event

The event is placed on the interrupt event pending queue. On exit from the interrupt path the Kernel processes all events on the interrupt pending queue as described in (a) above. This means the normal asynchronous event routines are called in an extension of normal (non-interrupt) processing between interrupt return and the main program. The routine is, therefore, not subject to the restrictions imposed on interrupt path routines.

c. In the Interrupt Path - Express Asynchronous Event

The event routine is called immediately, in the interrupt path. The routine must be at a 'near address' (see KL INIT EVENT). Under no circumstances may the routine enable interrupts.

KL EVENT enables interrupts unless it is called from the interrupt path.

Related entries:

KL INIT EVENT
KL NEXT SYNC
KL POLL SYNCHRONOUS
KL SYNC RESET

167: KL SYNC RESET #BCF5

Clear synchronous event queue.

Action:

The synchronous event queue is set empty - any outstanding events are simply discarded. The current event priority, used by KL POLL SYNCHRONOUS and KL NEXT SYNC to mask out lower priority events, is reset.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.

All other registers preserved.

Notes:

It is the user's responsibility to ensure that discarded events and any currently active events are reset. The event count of discarded events will be greater than zero, so any further 'kicks' will simply increment the count, but not add the event to the synchronous event queue - the events are, therefore, effectively disarmed.

Related entries:

KL DEL SYNCHRONOUS
KL NEXT SYNC
KL POLL SYNCHRONOUS

168: KL DEL SYNCHRONOUS #BCF8

Remove a synchronous event from the event queue.

Action:

The event is disarmed. If it is on the synchronous event queue then it is removed.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Deleting an event from the queue prevents the outstanding 'kicks' from being processed.

Before a synchronous event block is reset or reinitialized this routine should be used to ensure that it is not currently pending.

This routine enables interrupts.

Related entries:

KL DISARM EVENT
KL INIT EVENT
KL SYNC RESET

169: KL NEXT SYNC #BCFB

Get next event from the queue.

Action:

If there is an event on the synchronous event queue whose priority is greater than the current event priority (if any), then remove the event from the queue, set the current event priority to that of the event removed and return the previous event priority.

Entry conditions:

No conditions.

Exit conditions:

If there is an event to be processed:

Carry true.
HL contains the address of the event block.
A contains the previous event priority (if any).

If there is no event to be processed:

Carry false.
A and HL corrupt.

Always:

DE corrupt.
All other registers preserved.

Notes:

KL NEXT SYNC returns the address of the next event to be processed, if any, which it has taken off the synchronous event queue and whose priority has now been set as the event priority mask.

The foreground program should call KL POLL SYNCHRONOUS regularly to check for outstanding events. KL POLL SYNCHRONOUS is a short routine in RAM, so calling it imposes little overhead. If there is an event outstanding then the above procedure should be invoked, and should be repeated until the event queue is empty.

The current event priority mechanism allows event routine to poll for, and process, events of higher priority. The priority returned by this routine must be preserved until it is passed to KL DO SYNC.

KL NEXT SYNC enables interrupts.

The procedure for processing synchronous events is as follows:

```
TRY AGAIN:
CALL KL_NEXT_SYNC      ;return next event, if any
JR NC,??????           ;jump if no event to process
;
PUSH HL                ;save address of event
PUSH AF                ;save previous event priority
CALL KL_DO_SYNC        ;call the event routine
POP AF
POP HL
;
CALL KL_DONE_SYNC      ;reset the event priority mask, deal with
                       ;the event count and put the event back
                       ;on the queue if the count is still greater
                       ;than zero
JR TRY_AGAIN           ;see if any events are still awaiting processing
```

Related entries:

KL DONE SYNC
KL DO SYNC
KL EVENT
KL INIT EVENT
KL POLL SYNCHRONOUS

170: KL DO SYNC #BCFE

Perform an event routine.

Action:

Call the event routine for a given event.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine is intended to be called to process an event after KL NEXT SYNC has found it to be pending. Use of this entry at any other time is not recommended.

See KL NEXT SYNC above for the general scheme for processing synchronous events.

KL DO SYNC does not itself affect the event count.

Related entries:

KL DONE SYNC
KL NEXT SYNC

171: KL DONE SYNC #BD01

Finish processing an event.

Action:

Once a synchronous event has been processed, by invoking its event routine via KL DO SYNC, this entry must be called to restore the current event priority and to deal with the event count. If the count remains greater than zero the event block is placed back on the synchronous event queue.

Entry conditions:

A contains the previous event priority.

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

This routine is intended to be called after calling KL NEXT SYNC, to find a pending event, and KL DO SYNC, to run the event routine. It uses the previous event priority and the event block address returned by KL NEXT SYNC. Other uses of this entry are not recommended.

See KL NEXT SYNC above for the general scheme for processing synchronous events.

Restoring the current event priority is an essential step in maintaining the synchronous event priority scheme.

If the event count is greater than zero then it is decremented. If the count is still greater than zero then there are further events outstanding and the event is placed back on the synchronous event queue. The event may be disarmed between KL NEXT SYNC and KL DONE SYNC. Setting the event count to one before calling KL DONE SYNC forces multiple events to be treated as a single event.

KL DONE SYNC may enable interrupts.

Related entries:

KL DO SYNC
KL NEXT SYNC

172: KL EVENT DISABLE #BD04

Disable normal synchronous events.

Action:

Prevent normal synchronous events from being processed but allow express synchronous events to be processed. This is achieved by setting the current event priority higher than any possible normal synchronous event priority.

Entry conditions:

No conditions.

Exit conditions:

HL corrupt.

All other registers preserved.

Notes:

KL EVENT DISABLE does not prevent events from being kicked. The effect is to 'mask off' all pending normal synchronous events so that they are hidden from the foreground program (when KL POLL SYNCHRONOUS or KL NEXT SYNC are called) and hence are not processed.

KL EVENT ENABLE reverses the effect of KL EVENT DISABLE.

It is not possible to disable synchronous events permanently from inside a synchronous event routine as the previous current event priority is restored when the event routine returns.

Related entries:

KL DISARM EVENT
KL EVENT ENABLE
KL NEXT SYNC
KL POLL SYNCHRONOUS

173: KL EVENT ENABLE #BD07

Enables normal synchronous events.

Action:

Allow normal and express synchronous events to be processed.

Entry conditions:

No conditions.

Exit conditions:

HL corrupt.

All other registers preserved.

Notes:

Events are enabled by default. KL EVENT ENABLE reverses the effect of KL EVENT DISABLE.

It is not possible to disable synchronous events permanently from inside a synchronous event routine as the current event priority which is used to disable events is restored when the event routine returns.

Related entries:

KL EVENT DISABLE
KL NEXT SYNC
KL POLL SYNCHRONOUS

174: KL DISARM EVENT #BD0A

Prevent an event from occurring.

Action:

Disarms the event by setting the event count to a negative value. Any further 'kicks' (calls of KL EVENT) for the event will be ignored, any outstanding events are discarded.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

KL DISARM EVENT should only be used with asynchronous events. Synchronous events may be disarmed by calling KL DEL SYNCHRONOUS, which also ensures that the event is not on the synchronous event queue.

The event may be rearmed by reinitializing it (KL INIT EVENT) or by setting the event count (byte 2 of the event block) to zero.

Related entries:

KL DEL SYNCHRONOUS
KL INIT EVENT

175: KL TIME PLEASE #BD0D

Ask the elapsed time.

Action:

The Kernel maintains a count which it increments on each time interrupt. The count, therefore, measures time in 1/300th of a second units. This routine returns the current count.

Entry conditions:

No conditions.

Exit conditions:

DEHL contains the four byte count (D contains the most significant byte and L the least significant byte).

All other registers preserved.

Notes:

The count is zeroized when the machine is turned on or reset. The count may be set to another starting value by KL TIME SET.

The count is not kept up to date if interrupts are disabled for long periods, such as while reading and writing the cassette.

The four byte count overflows after approximately:

14,316,558 Seconds
= 238,609 Minutes
= 3,977 Hours
= 166 Days

This routine enables interrupts.

Related entries:

KL TIME SET

176: KL TIME SET #BD10

Set the elapsed time.

Action:

The Kernel maintains a count which it increments on each time interrupt. The count, therefore, measures time in 1/300th of a second units. This routine sets the count to a given value.

Entry conditions:

DEHL contains the four byte count to set (D contains the most significant byte and L the least significant byte).

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The four byte count overflows after approximately:

14,316,558 Seconds
= 238,609 Minutes
= 3,977 Hours
= 166 Days

KL TIME SET may be used to set the count to the actual time of day, so that the Kernel then maintains a real clock rather than a simple measure of the time elapsed since the last reset.

The count is not kept up to date if interrupts are disabled for long periods, such as while reading and writing the cassette.

This routine enables interrupts.

Related entries:

KL TIME PLEASE

177: MC BOOT PROGRAM #BD13

Load and run a program.

Action:

Shut down as much of the system as possible then load a program into RAM and run it. If the load fails then the previous foreground program is restarted.

Entry conditions:

HL contains the address of the routine to call to load the program.

Exit conditions:

Does not exit!

Notes:

The system is partially reset before attempting to load the program. External interrupts are disabled, as are all timer, frame flyback and keyboard break events. Sound generation is turned off, indirections are set to their default routines and the stack is reset to the default system stack. This process ensures that no memory outside the firmware variables area is in use when loading the program. Overwriting an active event block or indirection routine could otherwise have unfortunate consequences.

The partial system reset does not change the ROM state or ROM selection. The routine run to load the program must be in accessible RAM or an enabled ROM. Note that the firmware jumpblock normally enables the lower ROM and disables the upper ROM and so the routine must normally be in RAM above #4000 or in the lower ROM.

The routine run to load the program is free to use any store from #0040 up to the base of the firmware variables area (#B100) and may alter indirections and arm external device interrupts as required. It should obey the following

Exit conditions:

If the program loaded successfully:

Carry true.
HL contains the program entry point.

If the program failed to load:

Carry false.
HL corrupt.

Always:
A, BC, DE, IX, IY and other flags corrupt.

After a successful load the firmware is completely initialized (as at EMS) and the program is entered at the entry address returned by the load routine. Returning from the program will reset the system (perform RST 0).

After an unsuccessful load an appropriate error message is printed and the previous foreground program is restarted. If the previous foreground program was itself a RAM program then the default ROM is entered instead as the program may have been corrupted during the failed loading.

Related entries:

CAS IN DIRECT
KL CHOKE OFF
MC START PROGRAM

178: MC START PROGRAM #BD16

Run a foreground program.

Action:

Fully initialize the system and enter a program.

Entry conditions:

HL contains the entry point address.

C contains the required ROM selection.

Exit conditions:

Never exits!

Notes:

HL and C comprise the 'far address' of the entry point of the foreground program (see section 2).

When entering a foreground program in ROM the ROM selection should be that required to select the appropriate ROM. When entering a foreground program in RAM the ROM selection should be used to enable or disable ROMs as the RAM program requires (ROM select addresses #FC..#FF).

This routine carries out a full EMS initialization of the firmware before entering the program. Returning from the program will reset the system (perform RST 0).

MC START PROGRAM is intended for running programs in ROM or programs that have already been loaded into RAM. To load and run a RAM program use MC BOOT PROGRAM.

Related entries:

MC BOOT PROGRAM
RESET ENTRY (RST 0)

179: MC WAIT FLYBACK #BD19

Wait for frame flyback.

Action:

Wait until frame flyback occurs.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

Frame flyback is a signal generated by the CRT controller to signal the start of the vertical retrace period. During this period the screen is not being written and so major operations can be performed on the screen without producing unsightly effects. A prime example is rolling the screen.

The frame flyback signal only lasts for a couple of hundred microseconds but the vertical retrace period is much longer than this. However, there will be a ticker interrupt in the middle of frame flyback which may cause the foreground processing to be suspended for a significant length of time. It is important, therefore, to perform any critical processing as soon after the frame flyback is detected as is possible.

This routine returns immediately if frame flyback is occurring when it is called. It does not wait for the start of frame flyback (use a frame flyback event to do this).

Related entries:

KL ADD FRAME FLY

180: MC SET MODE #BD1C

Set the screen mode.

Action:

Load the hardware with the required screen mode.

Entry conditions:

A contains the required mode.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The required mode is checked and no action is taken if it is invalid. If it is valid then the new value is sent to the hardware.

The screen modes are:

0:	160 x 200 pixels,	20 x 25 characters.
1:	320 x 200 pixels,	40 x 25 characters.
2:	640 x 200 pixels,	80 x 25 characters.

Altering the screen mode without notifying the Screen Pack will produce peculiar effects on the screen. In general SCR SET MODE should be called to change screen mode. This, in turn, sets the new mode into the hardware.

Related entries:

SCR SET MODE

181: MC SCREEN OFFSET #BD1F

Set the screen offset.

Action:

Load the hardware with the offset of the first byte on the screen inside a 2K screen block and which 16K block the screen memory is located in.

Entry conditions:

A contains the new screen base.

HL contains the new screen offset.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The screen base address is masked with #C0 to make sure it refers to a valid 16K memory area. The default screen base is #C0 (the screen is underneath the upper ROM).

The screen offset is masked with #07FE to make it legal. Note that bit 0 is ignored as the hardware only uses even offsets.

If the screen base or offset is changed without notifying the Screen Pack then unexpected effects may occur on the screen. In general SCR SET BASE or SCR SET OFFSET should be called. These, in their turn, send the values to the hardware.

Related entries:

SCR SET BASE
SCR SET OFFSET

182: MC CLEAR INKS #BD22

Set all inks to one colour.

Action:

Set the colour of the border and set the colour of all the inks. All inks are set to the same colour thus giving the impression that the screen has been cleared instantly.

Entry conditions:

DE contains the address of an ink vector.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

The ink vector has the form:

Byte 0:	Colour of the border.
Byte 1:	Colour for all inks.

The colours supplied are the numbers used by the hardware rather than the grey scale numbers supplied to SCR SET INK (see Appendix V).

After the screen has been cleared (or whatever) the correct ink colours can be set by calling MC SET INKS.

This routine sets the colours for all 16 inks whether they can be displayed on the screen in the current mode or not.

This ink clearing technique is used by the Screen Pack when clearing the screen or changing mode (by SCR CLEAR and SCR SET MODE).

Related entries:

MC SET INKS

183: MC SET INKS #BD25

Set colours of all the inks.

Action:

Set the colours of all the inks and the border.

Entry conditions:

DE contains the address of an ink vector.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The ink vector passed has the following layout:

Byte 0:	Colour of the border.
Byte 1:	Colour for ink 0.
Byte 2:	Colour for ink 1.
…	...
Byte 16:	Colour for ink 15.

The colours supplied are the numbers used by the hardware rather than the grey scale numbers supplied to SCR SET INK (see Appendix V).

This routine sets the colours for all inks including those that cannot be visible in the current screen mode. However, it is only necessary to supply sensible colours for the visible inks.

The Screen Pack sets the colours for all the inks each time the inks flash and after an ink colour has been changed (by calling SCR SET INK or SCR SET BORDER).

Related entries:

MC CLEAR INKS
SCR SET BORDER
SCR SET INK

184: MC RESET PRINTER #BD28

Reset the printer indirection.

Action:

Set the printer indirection, MC WAIT PRINTER, to its default routine and, in V1.1 firmware, set up the default printer translation table.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

The default printer translation table is described in Appendix XIV. This is designed to drive the DMP-1 printer. It only translates the additional characters in the character set (#A0..#AF); it does not translate any of the standard ASCII characters or the graphics characters.

Related entries:

MC WAIT PRINTER
MC PRINT CHAR

185: MC PRINT CHAR #BD2B

Try to send a character to the Centronics port.

Action:

Send a character to the printer (Centronics port) or time out if the printer is busy for too long.

Entry conditions:

A contains the character to send (bit 7 is ignored).

Exit conditions:

If the character was sent OK:

Carry true.

If the printer timed out:

Carry false.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

In V1.1 firmware, the character to be sent is translated using the printer translation table as set by MC PRINT TRANSLATION. If the supplied character is not found in the table then it is sent as supplied without translation. However, if the character is found in the translation table then the corresponding translation is sent instead; unless the translation is FF in which case the character is ignored and nothing is sent.

This routine calls the Machine Pack indirection MC WAIT PRINTER to sent the character. The default indirection routine waits for the Centronics port to become non-busy then sends the character. If the port remains busy for too long (approximately 0.4 seconds) then the routine times out and the character is not sent. This time out is provided so that the caller can test for break whilst driving the printer.

Related entries:

MC PRINT TRANSLATION
MC RESET PRINTER
MC WAIT PRINTER

186: MC BUSY PRINTER #BD2E

Test if the Centronics port is busy.

Action:

Test if the printer (Centronics port) is busy.

Entry conditions:

No conditions.

Exit conditions:

If Centronics port is busy:
Carry true.

If Centronics port is idle:
Carry false.

Always:
Other flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

MC SEND PRINTER

187: MC SEND PRINTER #BD31

Send a character to the Centronics port.

Action:

Send a character to the printer (Centronics port) which must not be busy.

Entry conditions:

A contains the character to send (bit 7 is ignored).

Exit conditions:

Carry true.

A and other flags corrupt.
All other registers preserved.

Notes:

The printer must not be busy when a character is sent. The higher level routine MC PRINT CHAR will automatically wait for the printer to become non-busy and should be used in preference.

Related entries:

MC BUSY PRINTER
MC PRINT CHAR

188: MC SOUND REGISTER #BD34

Send data to a sound chip register.

Action:

Set a sound chip register. This is a rather convoluted action because of the way the hardware has been designed.

Entry conditions:

A contains the sound chip register number.

C contains the data to send.

Exit conditions:

AF and BC corrupt.

All other registers preserved.

Notes:

This routine enables interrupts.

Related entries:

None!

189: JUMP RESTORE #BD37

Restore the standard jumpblock.

Action:

Set the main firmware jumpblock to its standard state as described in sections 14.1 and 15.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

This routine may be used to restore the jumpblock to its standard routines after the user has changed entries in it. The whole of the jumpblock is set up so care must be taken if other programs, such as AMSDOS, have patched it.

The indirections jumpblock is set up piecemeal by the various packs' initialization and reset routines. JUMP RESTORE does not set up the indirections.

Related entries:

GRA RESET
KM RESET
MC RESET PRINTER
SCR RESET
TXT RESET

190: KM SET LOCKS #BD3A

Set the shift and caps lock states.

Action:

Turn the shift and caps locks on or off.

Entry conditions:

H contains the required caps lock state.
L contains the required shift lock state.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

The lock states are:

#00	means that the lock is to be turned off.
#FF	means that the lock is to be turned on.

The default lock states are off.

Related entries:

KM GET STATE

191: KM FLUSH #BD3D

Flush the keyboard buffers.

Action:

Discard all pending keys from the key buffer, the 'put back' character and any current expansion string.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.

All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

The next character that will be returned by KM READ CHAR (or a similar routine) after KM FLUSH is called will be the first character that the user types after the call of KM FLUSH since all the pending characters will have been discarded.

On V1.0 firmware the effect of this routine can be simulated by repeatedly calling KM READ CHAR until it comes back with carry false.

Related entries:

KM READ CHAR
KM READ KEY

192: TXT ASK STATE #BD40

Get the state of the Text VDU.

Action:

Returns the VDU enable/disable state and the cursor on/off and cursor enable/disable states of the current selected stream.

Entry conditions:

No conditions.

Exit conditions:

A contains the stream state.

Flags corrupt.
All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

The stream state is returned as follows:

Bit 0	0 → cursor enabled,	1 → cursor disabled.
Bit 1	0 → cursor on,	1 → cursor off.
Bits 2..6	are undefined.	
Bit 7	0 → VDU disabled,	1 → VDU enabled.

Related entries:

TXT CUR DISABLE
TXT CUR ENABLE
TXT CUR OFF
TXT CUR ON
TXT VDU DISABLE
TXT VDU ENABLE

193: GRA DEFAULT #BD43

Set the default Graphics VDU modes.

Action:

Sets the graphics write mode, background mode, first pixel mode and line mask to their default settings.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.

All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

This routine sets the following modes:

Graphics write mode is set to force.
Graphics background mode is set to opaque.
First point mode is set to plot the first pixel of lines.
Line mask is set to give continuous lines (mask of #FF).

Related entries:

GRA INITIALISE
GRA RESET
GRA SET BACK
GRA SET FIRST
GRA SET LINE MASK
SCR ACCESS

194: GRA SET BACK #BD46

Set whether background is to be written.

Action:

Set the graphics background write mode to opaque or transparent. This affects how GRA LINE ABSOLUTE, GRA LINE RELATIVE and GRA WR CHAR write 'background' pixels. In opaque mode the pixels are written in the current paper ink using the current graphics write mode. In transparent mode these pixels are not plotted at all.

Entry conditions:

If background is to be written (opaque mode):
A must be zero.

If background is not to be written (transparent mode):
A must be non-zero.

Exit conditions:

All registers and flags preserved.

Notes:

This routine is not available on V1.0 firmware.

Transparent write mode is useful for annotating diagrams and for similar applications.

The graphics background write mode is similar to but independent of the character write mode of each stream of the Text VDU.

The default setting is opaque mode.

Related entries:

GRA DEFAULT
GRA LINE
GRA LINE ABSOLUTE
GRA LINE RELATIVE
GRA SET LINE MASK
GRA WR CHAR
TXT SET BACK

195: GRA SET FIRST #BD49

Set whether the first point of a line is to be plotted.

Action:

Turn plotting of the first pixel of lines on or off.

Entry conditions:

If the first pixel is not to be plotted:
  A contains zero.

If the first pixel is to be plotted:
  A contains non-zero.

Exit conditions:

All registers and flags preserved.

Notes:

This routine is not available on V1.0 firmware.

Turning off the plotting of the first pixel of a line is particularly useful when drawing using XOR graphics write mode. For example, if a box is drawn in XOR mode when the first pixel of lines are being plotted then the corner pixels will e plotted twice and will therefore not be set. By not plotting the first pixel of lines this effect is avoided.

The default setting for this mode is to plot the first pixel.

Related entries:

GRA DEFAULT
GRA LINE
GRA LINE ABSOLUTE
GRA LINE RELATIVE

196: GRA SET LINE MASK #BD4C

Set the line mask for plotting pixels of lines.

Action:

Set the line mask that specifies how pixels on lines are to be plotted. Where a bit in the mask is set the pixel will be plotted in the foreground (in graphics pen ink using the graphics write mode). Where a pixel in the mask is not set the pixel will either be plotted in the graphics paper ink using the graphics write mode or will not be plotted at all depending on the graphics background write mode.

Entry conditions:

A contains the line mask to use.

Exit conditions:

All registers and flags preserved.

Notes:

This routine is not available on V1.0 firmware.

The line mask is used starting with bit 7 and running to bit 0 and then starting with bit 7 again. Successive lines will use the mask as it was left when the previous line finished, the mask is not reset between lines.

The line mask specifies how pixels are to be plotted. This means that the same mask will give noticeably different effects in the various screen modes.

The mask is applied to the line running from left to right or from bottom to top, depending on the angle of the line, irrespective of which way round the end points of the line are specified.

If the first pixel of the line is not being plotted then the line mask is applied to the second pixel of the line first. It is not stepped on for the missing first pixel.

The default line mask is #FF which plot the whole line in the foreground.

Related entries:

GRA DEFAULT
GRA LINE
GRA LINE ABSOLUTE
GRA LINE RELATIVE
GRA SET BACK

197: GRA FROM USER #BD4F

Convert user coordinates to base coordinates.

Action:

Convert the coordinates of a point from user coordinates to base coordinates rounding as appropriate.

Entry conditions:

DE contains the user X coordinate.
HL contains the user Y coordinate.

Exit conditions:

DE contains the base X coordinate.
HL contains the base Y coordinate.

AF corrupt.
All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

The following formulae are used to convert between the coordinate systems:

Base X = (Origin X + Rounded X) / Points per pixel
Rounded X = (User X + Round factor) AND Round mask

Where:

			Round mask	Points per Pixel
	Round factor			
	+ve user X	-ve user X		
Mode 0:	0	0	#FFFF	1
Mode 1:	0	1	#FFFE	2
Mode 2:	0	3	#FFFC	4

Base Y = (Origin Y + Rounded Y) / Points per pixel
Rounded Y = (User Y + Round factor) AND Round mask

Where:

Round factor = 0 for +ve user Y
= 1 for -ve user Y
Round mask = #FFFE
Points per pixel = 2

This routine is particularly useful when calling Screen pack routines which take the positions of points in base coordinates.

Related entries:

GRA SET ORIGIN
SCR DOT POSITION

198: GRA FILL #BD52

Fill an area of the screen.

Action:

Fill an area of the screen containing the current graphics position and bounded by the edge of the window and pixels set to the pen ink.

Entry conditions:

A contains the (unencoded) ink to fill the area with.

HL contains the address of a buffer.

DE contains the length of the buffer.

Exit conditions:

If the area was filled successfully:

Carry true.

If the area was not filled:

Carry false.

Always:

A, BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

The filling algorithm treats pixels set to the current pen ink and pixels set to the ink that is being used for filling as delimiters of the edge of the area. The fill ink and the pen ink may be the same ink.

Pixels that are filled are set to the fill ink. The graphics write mode does not affect the way pixels are written when filling.

The filling algorithm only moves up, down, right or left. It does not move diagonally and so the algorithm will not 'escape' through a gap between the edge pixels that are diagonally adjacent. This means that the edge can be delimited using the normal lines drawn by the Graphics VDU.

The filling algorithm avoids recursing. Instead it stores 'interesting points', places that the algorithm has chosen one route to fill but might have chosen another route, in the buffer supplied by the user. The buffer may lie anywhere in RAM. Each 'interesting point' stored uses 7 bytes of the buffer and there is an overhead of 1 byte used to mark the end of the buffer. Thus a buffer 64 bytes long will allow 9 'interesting points' to be stored which should be sufficient for filling most simple areas.

The area to be filled may be as complicated as required but the more complicated the shape the longer the 'interesting point' buffer needs to be.

The failure return from this routine can occur for three reasons. Firstly, the current graphics position may be outside the window. Secondly, the pixel at the current graphics position may be edge (pen or fill ink). In these cases the routine will return without filling anything. Thirdly the algorithm may exhaust the 'interesting point' buffer in which case some portion of the area will not be filled.

Related entries:

GRA SET PEN

199: SCR SET POSITION #BD55

Set the location of the screen memory.

Action:

Tell the Screen pack the screen base and the offset without telling the hardware.

Entry conditions:

A contains the screen base.
HL contains the screen offset.

Exit conditions:

A contains the screen base masked as required.
HL contains the screen offset masked as required.

Flags corrupt.
All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

This routine changes the location of the screen without notifying the hardware of the change. This effect may be used to construct a second screen of text or graphics without clearing the previous screen. When the new screen has been constructed the hardware may be notified and the picture will appear instantly.

In general the user is advised to set the base using SCR SET BASE and the offset using SCR SET OFFSET.

The screen base is masked with #C0 and the screen offset with #07FE to make the values legal.

Related entries:

SCR GET LOCATION
SCR SET BASE
SCR SET OFFSET

200: MC PRINT TRANSLATION #BD58

Set the printer translation table.

Action:

Set how characters are to be translated before being sent to the printer.

Entry conditions:

HL contains the address of the table.

Exit conditions:

If the table is too long (more than 20 entries):
  Carry false.

If the table is OK:
  Carry true.

Always:
  A, BC, DE, HL and other flags corrupt.
  All other registers preserved.

Notes:

This routine is not available on V1.0 firmware.

The supplied translation table may lie anywhere in RAM. This routine copies the table and so the memory may be re-used if required.

The format of the table is as follows:

Byte 0:	Number of entries in the table (N).
Bytes 1,2:	Entry 1
…	...
Bytes 2N-1,2N:	Entry N

The format of each two byte entry is as follows:

Byte 0:	Character to be translated.
Byte 1:	Character to translate to.

If the character to translate to is #FF then the character is ignored and nothing is sent to the printer. Translation of characters by the printer driver occurs in MC PRINT CHAR.

The default translation table is set up when MC RESET PRINTER is called. The default table is designed to drive the DMP-1 printer (see Appendix XIV).

Related entries:

MC PRINT CHAR

201: KL BANK SWITCH #BD5B

Select a memory organization.

Action:

Set which RAM banks are switched into the 64K of RAM in the memory map.

Entry conditions:

A contains new organization.

Exit conditions:

A contains old organization.

Flags corrupt.
All other registers preserved.

Notes:

This routine is only available on the CPC6128 (i.e. V1.2 firmware). The memory organizations and bank switching are discussed fully in section 2.5.

It is inadvisable to bank switch to a memory organization where the code that is being executed, or stack are inaccessible!

Related entries:

KL L ROM DISABLE
KL L ROM ENABLE
KL ROM SELECT
KL U ROM DISABLE
KL U ROM ENABLE