# QPC
## Concepts

The Concepts Reference Guide describes concepts relating to SBASIC and QPC. It is best to think of the Concept Guide as a source of information. If there are any questions about SBASIC or QPC itself which arise out of using the emulator or other sections of the manual then the Concept Guide may have the answer. Concepts are listed in alphabetical order using the most likely term for that concept. If the subject cannot be found then consult the index which should be able to tell you which page to turn to.

Where an example is listed with line numbers, then it is a complete program and can be entered and run. Examples listed without line numbers are usually simple commands and it may not always be sensible to enter them into the emulator in isolation.

This guide is a combination of the Sinclair QL manuals Concepts section, the (Super)Gold card manual, the Toolkit 2 manual, the QPAC2 (Extended Environment), the SMSQ/E manual, and the QPC manual.

## arrays

Arrays must be **DIM**ensioned before they are used. When an array is dimensioned the value of each of its elements is set to zero or a zero length string if it is a string array. An array dimension runs from zero up to the specified value. There is no limits to the number of dimensions which can be defined other than the total memory capacity of the computer. An array of data is stored such that the last index defined cycles round most rapidly:

the array defined by

**DIM array(2,4)**

will be stored as

0,0 low address
0,1
0,2
0,3
0,4
1,0
1,1
1,3
1,4
2,0
2,1
2,2
2,3
2,4 high address

| Command | Function |
|---|---|
| **DIM** | dimension an array |
| **DIMN** | find out about the dimensions of an array |

# background

The screen background (the area beneath any open windows) can be set to a plain or stippled colour, or an image.

**BGCOLOUR_QL** sets the background to one of the QL mode colours, or stipple patterns. Background colours and stipples are defined in the same way as normal QL mode colours as used in **INK**, **PAPER** etc.

| | |
|---|---|
| **BGCOLOUR_QL 255** | Sets the background colour to a black and white check |
| **BGCOLOUR_QL 1** | Sets the background colour to blue |

**BGCOLOUR_24** sets the background colour to one of the 16 Million (24 Bit) true colours, The background colour is defined in the same way as normal 24 Bit colours as used in **INK**, **PAPER** etc.

| | |
|---|---|
| **BGCOLOUR_24 40** | Sets the background colour to deep blue |

**BGIMAGE** allows the use of a background image instead of a solid colour. The image which is stored in a file, is loaded by.

| | |
|---|---|
| **BGIMAGE win1_wallpaper** | Load a wallpaper |

Background images must be in the form of a screen snapshot at the screen resolution you are using. To create a background image.

```
100  WINDOW SCR_XLIM,SCR_YLIM,0,0  :  REMark Full screen
       .
       .  Draw the wallpaper on the screen
       .
900  SBYTES_O win1_wallpaper,SCR_BASE,SCR_LLEN * SCR_YSIZE
```

Generating a screen image in this way as a program, prevents the cursor and the save command spoiling the generated image.

| Command | Function |
|---|---|
| **BGCOLOUR_QL** | set background to a solid colour or stipple |
| **BGCOLOUR_24** | set background to a solid colour |
| **BGIMAGE** | load a background image |

# BASIC

SBASIC includes most of the functions, procedures and constructs found in other dialects of BASIC. Many of these functions are superfluous in SBASIC but are included for compatibility reasons:

| | |
|---|---|
| **GOTO** | use **IF**, **REPEAT**, etc |
| **GOSUB** | use **DEFine PROCedure** |
| **ON...GOTO** | use **SELect** |
| **ON...GOSUB** | use **SELect** |

Some commands appear not to be present. They can always be obtained by using a more general function. For example, there are no **LPRINT** or **LLIST** statements in SBASIC but output can be directed to a printer by opening the relevant channel and using **PRINT** or **LIST**.

| | |
|---|---|
| **LPRINT** | use **PRINT #** |
| **LLIST** | use **LIST #** |
| **VAL** | not required in SBASIC |
| **STR$** | not required in SBASIC |
| **IN** | not applicable to 68000 processor |
| **OUT** | not applicable to 68000 processor |

comment: Almost all forms of **BASIC** require the **VAL(x$)** and **STR$(x)** functions in order to be able to convert the internal codified form of the value of a string expression to or from the internal codified form of the value of a numeric expression.

These functions are redundant in SBASIC because of the provision of a unique facility referred to as "coercion". The **VAL** and **STR$** functions are therefore not provided.

# break

If at any time the computer fails to respond or you wish to stop a SBASIC program or command then press

**[CTRL] [SPACE]**

A program broken into in this way can be restarted by using the **CONTINUE** command.

Screen output may be paused by pressing either **CTRL F5**, or the **ScrLock** key.

To switch between a full screen display and a window, press **SHIFT CTRL F12**

To perform a soft reset, (restart SMSQ)

**[CTRL] [SHIFT] [ALT] [TAB]**

To terminate QPC

**[CTRL] [SHIFT] [ScrLock]**

QPC can also be ended with the **QPC_EXIT** command from SBASIC, or the "**X**" (close) button on the Windows title bar.

# channels

A channel is a means by which data can be output to or input from a QPC device. Before a channel can be used it must first be activated (or opened) with the **OPEN** command. Certain channels should always be kept open: these are the default channels and allow simple communication with QPC via the keyboard and screen. When a channel is no longer in use it can be deactivated (closed) with the **CLOSE** command.

A channel is identified by a channel number. A channel number is a numeric expression preceded by a #. When the channel is opened a device is linked to a channel number and the channel is initialised. Thereafter the channel is identified only by its channel number. For example:

```
OPEN #5,SER1
```

Will link serial port 1 to the channel number 5. When a channel is closed only the channel number need be specified. For example:

```
CLOSE #5
```

Opening a channel requires that the device driver for that channel be activated. Usually there is more than one way in which the device driver can be activated. This extra information is appended to the device name and passed to the **OPEN** command as a parameter. See concepts *device*.

Data can be output to a channel by **PRINT**ing to that channel; this is the same mechanism by which output appears on the QPC screen. **PRINT** without a parameter outputs to the default channel #1. For example:

```
10 OPEN #5,flp1_test_file
20 PRINT #5,"this text is in file test_file"
30 CLOSE #5
```

will output the text "this text is in file test_file" to the file test_file. It is important to close the file after all the accesses have been completed to ensure that all the data is written.

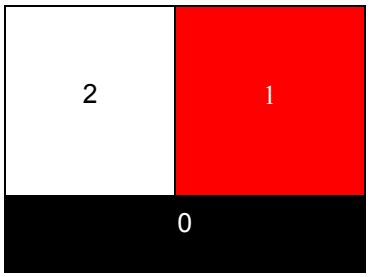
Data can be input from a file in an analogous way using **INPUT**. Data can be input from a channel a character at a time using **INKEY$**

**GET**, **PUT**, and their variations can also be used to output and input from a channel. **GET** and **PUT** allows data to be sent and read from a channel with more control than simply **PRINT**ing and **INPUT**ing.

A channel can be opened as a console channel; output is directed to a specified window on the QPC screen and input is taken from the QPC keyboard. When a console channel is opened the size and shape of the initial window is specified. If more than one console channel is active then it is possible for more than one channel to be requesting input at the same time. In this case, the required channel can be selected by pressing CTRL C to cycle round the waiting channels. The cursor in the window of the selected channel will flash.

QPC has three default channels which are opened automatically. Each of these channels is linked to a window on the QPC screen.

channel 0 - command and error channel
channel 1 - output and graphics channel
channel 2 - program listing channel




| Command | Function |
|---|---|
| **OPEN** | open a channel for I/O |
| **CLOSE** | close a previously opened channel |
| **PRINT** | output to a channel |
| **INPUT** | input from a channel |
| **INKEY$** | input a character from a channel |
| **GET** | unformatted input from a channel |
| **PUT** | unformatted output to a channel |

## character set and keys

The cursor controls are not built in to the operating system: however, if these functions are to be provided by applications software, they should use the keys specified; also the specified keys should not normally be used for any other purpose.

| Decimal | Hex | Keying | Display/Function |
|---|---|---|---|
| 0 | 00 | CTRL £ | NULL |
| 1 | 01 | CTRL A | |
| 2 | 02 | CTRL B | |
| 3 | 03 | CTRL C | Change input channel (see note) |
| 4 | 04 | CTRL D | |
| 5 | 05 | CTRL E | |
| 6 | 06 | CTRL F | |
| 7 | 07 | CTRL G | |
| 8 | 08 | CTRL H | |
| 9 | 09 | TAB (CTRL I) | Next field |
| 10 | 0A | ENTER (CTRL J) | New line / Command entry |
| 11 | 0B | CTRL K | |
| 12 | 0C | CTRL L | |
| 13 | 0D | CTRL M | Enter |
| 14 | 0E | CTRL N | |
| 15 | 0F | CTRL O | |
| 16 | 10 | CTRL P | |
| 17 | 11 | CTRL Q | |
| 18 | 12 | CTRL R | |
| 19 | 13 | CTRL S | |
| 20 | 14 | CTRL T | |
| 21 | 15 | CTRL U | |
| 22 | 16 | CTRL V | |
| 23 | 17 | CTRL W | |
| 24 | 18 | CTRL X | |
| 25 | 19 | CTRL Y | |
| 26 | 1A | CTRL Z | |
| 27 | 1B | ESC (CTRL SHIFT \|) | Abort current level of command |
| 28 | 1C | | |
| 29 | 1D | CTRL SHIFT ] | |
| 30 | 1E | | |
| 31 | 1F | | |
| 32 | 20 | SPACE | |
| 33 | 21 | SHIFT 1 | ! |
| 34 | 22 | SHIFT 2 | " |
| 35 | 23 | # | # |
| 36 | 24 | SHIFT 4 | $ |
| 37 | 25 | SHIFT 5 | % |
| 38 | 26 | SHIFT 7 | & |
| 39 | 27 | ' | ' |
| 40 | 28 | SHIFT 9 | ( |
| 41 | 29 | SHIFT 0 | ) |
| 42 | 2A | SHIFT 8 / Prt Screen | * |
| 43 | 2B | SHIFT = | + |
| 44 | 2C | , | , |
| 45 | 2D | - | - |
| 46 | 2E | . | . |
| 47 | 2F | / | / |

| Decimal | Hex | Keying | Display/Function |
|---|---|---|---|
| 48 | 30 | 0 | 0 |
| 49 | 31 | 1 | 1 |
| 50 | 32 | 2 | 2 |
| 51 | 33 | 3 | 3 |
| 52 | 34 | 4 | 4 |
| 53 | 35 | 5 | 5 |
| 54 | 36 | 6 | 6 |
| 55 | 37 | 7 | 7 |
| 56 | 38 | 8 | 8 |
| 57 | 39 | 9 | 9 |
| 58 | 3A | SHIFT ; | : |
| 59 | 3B | ; | ; |
| 60 | 3C | SHIFT . | < |
| 61 | 3D | = | = |
| 62 | 3E | SHIFT . | > |
| 63 | 3F | SHIFT / | ? |
| 64 | 40 | SHIFT ' | @ |
| 65 | 41 | SHIFT A | A |
| 66 | 42 | SHIFT B | B |
| 67 | 43 | SHIFT C | C |
| 68 | 44 | SHIFT D | D |
| 69 | 45 | SHIFT E | E |
| 70 | 46 | SHIFT F | F |
| 71 | 47 | SHIFT G | G |
| 72 | 48 | SHIFT H | H |
| 73 | 49 | SHIFT I | I |
| 74 | 4A | SHIFT J | J |
| 75 | 4B | SHIFT K | K |
| 76 | 4C | SHIFT L | L |
| 77 | 4D | SHIFT M | M |
| 78 | 4E | SHIFT N | N |
| 79 | 4F | SHIFT O | O |
| 80 | 50 | SHIFT P | P |
| 81 | 51 | SHIFT Q | Q |
| 82 | 52 | SHIFT R | R |
| 83 | 53 | SHIFT S | S |
| 84 | 54 | SHIFT T | T |
| 85 | 55 | SHIFT U | U |
| 86 | 56 | SHIFT V | V |
| 87 | 57 | SHIFT W | W |
| 88 | 58 | SHIFT X | X |
| 89 | 59 | SHIFT Y | Y |
| 90 | 5A | SHIFT Z | Z |
| 91 | 5B | [ | [ |
| 92 | 5C | \ | \ |
| 93 | 5D | ] | ] |
| 94 | 5E | SHIFT 6 | ^ |
| 95 | 5F | SHIFT - | _ |

| Decimal | Hex | Keying | Display/Function |
|---|---|---|---|
| 96 | 60 | SHIFT 3 | £ |
| 97 | 61 | A | a |
| 98 | 62 | B | b |
| 99 | 63 | C | c |
| 100 | 64 | D | d |
| 101 | 65 | E | e |
| 102 | 66 | F | f |
| 103 | 67 | G | g |
| 104 | 68 | H | h |
| 105 | 69 | I | i |
| 106 | 6A | J | j |
| 107 | 6B | K | k |
| 108 | 6C | L | l |
| 109 | 6D | M | m |
| 110 | 6E | N | n |
| 111 | 6F | O | o |
| 112 | 70 | P | p |
| 113 | 71 | Q | q |
| 114 | 72 | R | r |
| 115 | 73 | S | s |
| 116 | 74 | T | t |
| 117 | 75 | U | u |
| 118 | 76 | V | v |
| 119 | 77 | W | w |
| 120 | 78 | X | x |
| 121 | 79 | Y | y |
| 122 | 7A | Z | z |
| 123 | 7B | SHIFT [ | { |
| 124 | 7C | SHIFT \ | \| |
| 125 | 7D | SHIFT ] | } |
| 126 | 7E | SHIFT # | ~ |
| 127 | 7F | SHIFT ESC | © |
| 128 | 80 | CTRL ESC | ä |
| 129 | 81 | CTRL SHIFT 1 | ã |
| 130 | 82 | CTRL SHIFT ' | å |
| 131 | 83 | CTRL SHIFT 3 | é |
| 132 | 84 | CTRL SHIFT 4 | ö |
| 133 | 85 | CTRL SHIFT 5 | õ |
| 134 | 86 | CTRL SHIFT 7 | ø |
| 135 | 87 | CTRL ' | ü |
| 136 | 88 | CTRL SHIFT 9 | ç |
| 137 | 89 | CTRL SHIFT 0 | ñ |
| 138 | 8A | CTRL SHIFT 8 | z |
| 139 | 8B | CTRL SHIFT = | œ |
| 140 | 8C | CTRL , | á |
| 141 | 8D | CTRL - | à |
| 142 | 8E | CTRL . | â |
| 143 | 8F | CTRL / | ë |

| Decimal | Hex | Keying | Display/Function | |
|---|---|---|---|---|
| 144 | 90 | CTRL 0 | è | |
| 145 | 91 | CTRL 1 | ê | |
| 146 | 92 | CTRL 2 | ï | |
| 147 | 93 | CTRL 3 | í | |
| 148 | 94 | CTRL 4 | ì | |
| 149 | 95 | CTRL 5 | î | |
| 150 | 96 | CTRL 6 | ó | |
| 151 | 97 | CTRL 7 | ò | |
| 152 | 98 | CTRL 8 | ô | |
| 153 | 99 | CTRL 9 | ú | |
| 154 | 9A | CTRL SHIFT ; | ù | |
| 155 | 9B | CTRL ; | û | |
| 156 | 9C | CTRL SHIFT , | ß | |
| 157 | 9D | CTRL = | ¢ | |
| 158 | 9E | CTRL SHIFT . | ¥ | |
| 159 | 9F | CTRL SHIFT / | ` | |
| 160 | A0 | CTRL SHIFT 2 | Ä | |
| 161 | A1 | CTRL SHIFT A | Ã | |
| 162 | A2 | CTRL SHIFT B | Å | |
| 163 | A3 | CTRL SHIFT C | É | |
| 164 | A4 | CTRL SHIFT D | Ö | |
| 165 | A5 | CTRL SHIFT E | Õ | |
| 166 | A6 | CTRL SHIFT F | Ø | |
| 167 | A7 | CTRL SHIFT G | Ü | |
| 168 | A8 | CTRL SHIFT H | Ç¨ | |
| 169 | A9 | CTRL SHIFT I | Ñ | |
| 170 | AA | CTRL SHIFT J | Æ | |
| 171 | AB | CTRL SHIFT K | Œ | |
| 172 | AC | CTRL SHIFT L | α | alpha |
| 173 | AD | CTRL SHIFT M | δ | delta |
| 174 | AE | CTRL SHIFT N | θ | theta |
| 175 | AF | CTRL SHIFT O | λ | lambda |
| 176 | B0 | CTRL SHIFT P | μ | mu |
| 177 | B1 | CTRL SHIFT Q | π | pi |
| 178 | B2 | CTRL SHIFT R | ϕ | phi |
| 179 | B3 | CTRL SHIFT S | ¡ | |
| 180 | B4 | CTRL SHIFT T | ¿ | |
| 181 | B5 | CTRL SHIFT U | € | |
| 182 | B6 | CTRL SHIFT V | § | |
| 183 | B7 | CTRL SHIFT W | ¤ | |
| 184 | B8 | CTRL SHIFT X | « | |
| 185 | B9 | CTRL SHIFT Y | » | |
| 186 | BA | CTRL SHIFT Z | º | |
| 187 | BB | CTRL [ | ÷ | |
| 188 | BC | CTRL \ | ← | |
| 189 | BD | CTRL ] | → | |
| 190 | BE | CTRL SHIFT 6 | ↑ | |
| 191 | BF | CTRL SHIFT - | ↓ | |

| Decimal | Hex | Keying | Display/Function |
|---|---|---|---|
| 192 | C0 | Left | Cursor left one character |
| 193 | C1 | ALT Left | Cursor to start of line |
| 194 | C2 | CTRL Left / Backspace | Delete left one character |
| 195 | C3 | CTRL ALT Left | Delete line |
| 196 | C4 | SHIFT Left | Cursor left one word |
| 197 | C5 | SHIFT ALT Left | Pan left |
| 198 | C6 | SHIFT CTRL Left | Delete left one word |
| 199 | C7 | SHIFT CTRL ALT Left | |
| 200 | C8 | Right | Cursor right one character |
| 201 | C9 | ALT Right | Cursor to end of line |
| 202 | CA | CTRL Right / Delete | Delete character under cursor |
| 203 | CB | CTRL ALT Right | Delete to end of line |
| 204 | CC | SHIFT Right | Cursor right one word |
| 205 | CD | SHIFT ALT Right | Pan right |
| 206 | CE | SHIFT CTRL Right | Delete word under & right of cursor |
| 207 | CF | SHIFT CTRL ALT Right | |
| 208 | D0 | Up | Cursor right |
| 209 | D1 | ALT Up | Scroll up |
| 210 | D2 | CTRL Up | Search backward |
| 211 | D3 | ALT CTRL Up | |
| 212 | D4 | SHIFT Up / Page Up | Top of screen |
| 213 | D5 | SHIFT ALT Up / Home | |
| 214 | D6 | SHIFT CTRL Up | |
| 215 | D7 | SHIFT CTRL ALT Up | |
| 216 | D8 | Down | Cursor down |
| 217 | D9 | ALT Down | Scroll down |
| 218 | DA | CTRL Down | Search forwards |
| 219 | DB | ALT CTRL Down | |
| 220 | DC | SHIFT Down / Page Down | Bottom of screen |
| 221 | DD | SHIFT ALT Down / End | |
| 222 | DE | SHIFT CTRL Down | |
| 223 | DF | SHIFT CTRL ALT Down | |
| 224 | E0 | CAPS LOCK | Toggle CAPS LOCK function |
| 225 | E1 | ALT CAPS LOCK | |
| 226 | E2 | CTRL CAPS LOCK | |
| 227 | E3 | ALT CTRL CAPS LOCK | |
| 228 | E4 | SHIFT CAPS LOCK | |
| 229 | E5 | SHIFT ALT CAPS LOCK | |
| 230 | E6 | SHIFT CTRL CAPS LOCK | |
| 231 | E7 | SHIFT CTRL ALT CAPS LOCK | |
| 232 | E8 | F1 | |
| 233 | E9 | CTRL F1 | |
| 234 | EA | SHIFT F1 / F6 | |
| 235 | EB | CTRL SHIFT F1 | |
| 236 | EC | F2 | |
| 237 | ED | CTRL F2 | |
| 238 | EE | SHIFT F2 /F7 | |
| 239 | EF | CTRL SHIFT F2 | |

| Decimal | Hex | Keying | Display/Function |
|---|---|---|---|
| 240 | F0 | F3 | |
| 241 | F1 | CTRL F3 | |
| 242 | F2 | SHIFT F3 / F8 | |
| 243 | F3 | CTRL SHIFT F3 | |
| 244 | F4 | F4 | |
| 245 | F5 | CTRL F4 | |
| 246 | F6 | SHIFT F4 / F9 | |
| 247 | F7 | CTRL SHIFT F4 | |
| 248 | F8 | F5 | |
| 249 | F9 | CTRL F5 | |
| 250 | FA | SHIFT F5 / F10 | |
| 251 | FB | CTRL SHIFT F5 | |
| 252 | FC | SHIFT space / Insert | "Special" space |
| 253 | FD | SHIFT TAB | Back tab (CTRL ignored) |
| 254 | FE | SHIFT ENTER | "Special" newline (CTRL ignored) |
| 255 | FF | See below | |

Codes up to 20 hex are either control characters or non-printing characters. Alternative keyings are shown in brackets after the main keying.

Note that CTRL-C is trapped by SMSQ and cannot be detected without changes to the system variables.

Note that codes C0-DF are cursor control commands.

The ALT key depressed with any key combination other than cursor keys or CAPS LOCK generates the code FF, followed by a byte indicating what the keycode would have been if ALT had not been depressed.

Note that CAPS LOCK and CTRL-F5 are trapped by SMSQ and cannot be detected without special software.

# clock

SMSQ/E contains a real time clock, which runs when QPC is started. It obtains the current date and time from the Windows operating system on the PC. The SMSQ/E clock is then updated once per minute. So that the SMSQ/E clock, should never be more than one minute different from the Windows clock.

The format used for the date and time is standard ISO format.

**2001 JAN 01 12:09:10**

Individual year, month, day and time can all be obtained by assigning the string returned by DATE to a string variable and slicing it. The clock will run from 1961 JAN 01 00:00:00

Comment: For a description of the format, see BS5249: Part 1: 1976 and as modified in Appendix D.2.1 Table 5 Serial 5 and Appendix E.2 Table 6 Serials 1 and 2.

| Command | Function |
|---|---|
| **SDATE** | set the clock |
| **ADATE** | adjust the clock |
| **DATE** | return the date as a number |
| **DATE$** | return the date as a string |
| **DAY$** | return day of the week |
| **ALARM** | set an alarm |

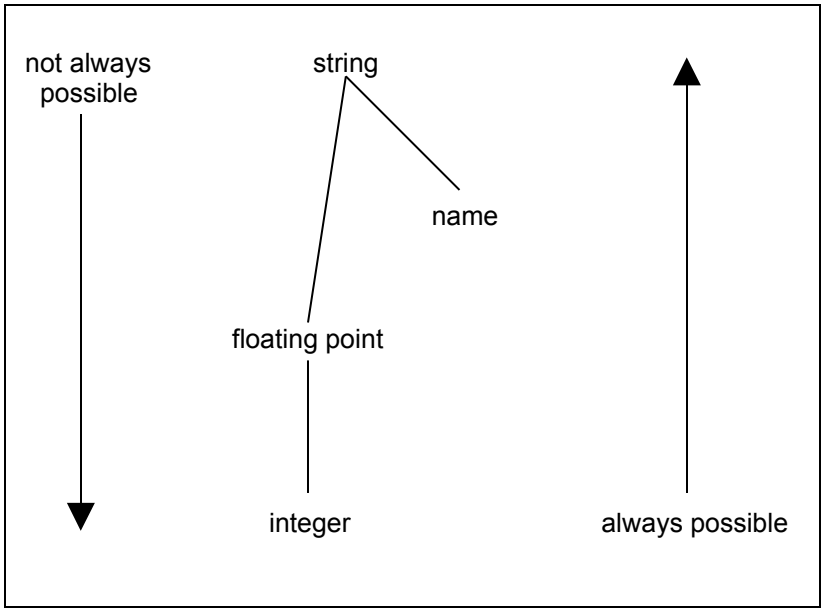
## coercion

If necessary SBASIC will convert the type of unsuitable data to a type which will allow the specified operation to proceed.

The operators used determine the conversion required. For example, if an operation requires a string parameter and a numeric parameter is supplied then SBASIC will first convert the parameter to type string. It is not always possible to convert data to the required form and if the data cannot be converted an error is reported.

The type of a function or procedure parameter can also be converted to the correct type. For example, the SBASIC **LOAD** command requires a parameter of type *name* but can accept a parameter of type *string* and which will be converted to the correct type by the procedure itself. Coercion of this form is always dependent on the way the function or procedure was implemented.

There is a natural ordering of data types in SMSQ/E, see figure below. String is the most general type since it can represent integer data (almost exactly). The figure below shows the ordering diagramatically. Data can always be converted moving up the diagram but it is not always possible moving down.




example: **a = b + c** (no conversion is necessary before performing the addition. Conversion is not necessary before assigning the result to a.)

**a% = b + c** (no conversion is necessary before performing the addition but the result is converted to integer before assigning.)

**a$ = b$ + c$** (b$ and c$ are converted to floating point, if possible, before being added together. The result is converted to string before assigning.)

**LOAD "flp1_data"** (the string **"flp1_data"** is converted to type name by the load procedure before it is used.)

comment: Statements can be written in SBASIC which would generate errors in most other computer languages. In general, it is possible to mix data types in a very flexible manner:

i. **PRINT "1" + 2 + "3"**
ii. **LET a$ = 1 + 2 + a$ + "4"**

# colour

QPC can operate in 4 different colour modes. Each executing job or SBASIC job may have it's own colour mode.

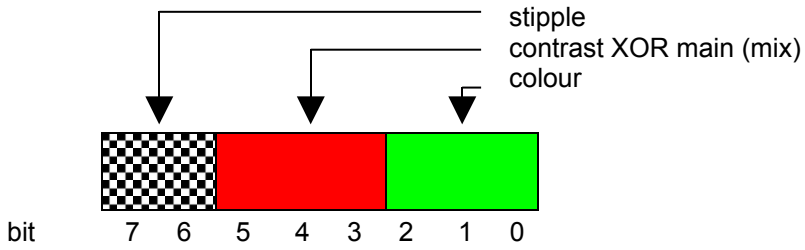
## COLOUR_QL

This is an 8 colour mode which can display either a **solid colour** or a **stipple** - a mixture of two colours to some predefined pattern. Colour specification in the **COLOUR_QL** mode, can be up to three items: a colour, a contrast colour and a stipple pattern.

When an SBASIC program starts executing, it is set to QL colour definition.

**single** *colour* := *composite_colour*

The single argument specifies the three parts of the colour specification. The main colour is contained in the bottom three bits of the colour byte. The next three bits contain the exclusive or (XOR) of the main colour and the contrast colour. The top two bits indicate the stipple pattern.




By specifying only the bottom three bits (i.e. the required colour) no *stipple* will be requested and a single solid colour will be used for display.

**double** *colour* := *background*, *contrast*

The *colour* is a *stipple* of the two specified colours. The default checkerboard stipple is assumed (stipple 3)

**triple** *colour* := *background*, *contrast*, *stipple*

*Background* and *contrast* colours and *stipple* are each defined separately.

**colours** The codes for standard palette colours:

| code | colour | bit pattern | composition | 24 bit value R | G | B |
|---|---|---|---|---|---|---|
| 0 | Black | 0 0 0 | | 00 | 00 | 00 |
| 1 | Blue | 0 0 1 | blue | 00 | 00 | FF |
| 2 | Red | 0 1 0 | red | FF | 00 | 00 |
| 3 | Magenta | 0 1 1 | red + blue | FF | 00 | FF |
| 4 | Green | 1 0 0 | green | 00 | FF | 00 |
| 5 | Cyan | 1 0 1 | green + blue | 00 | FF | FF |
| 6 | Yellow | 1 1 0 | green + red | FF | FF | 00 |
| 7 | White | 1 1 1 | green + red + blue | FF | FF | FF |

Colour Composition and Codes

**stipples** Stipples mix a background and a contrast colour in a fine stipple pattern. Stipples can be used in SMSQ/E in the same manner as ordinary solid colours. There are four stipple patterns:

**Stipple 0** **Stipple 1** **Stipple 2** **Stipple 3**

Stipple 3 is the default.

example: i. **PAPER 255 : CLS**
ii. **PAPER 2,4 : CLS**
iii. **PAPER 0,2,0 : CLS**

This program will display all of the colours and stipple patterns available in the **COLOUR_QL** mode.

```
100  REMark COLOUR_QL colours
110  WINDOW 750,550,25,25
120  COLOUR_QL
130  PAPER 0:INK 7
140  CLS
150  FOR x=0 TO 7
160   FOR y= 0 TO 31
170    PAPER 0:INK 7
180    AT y,15*x : PRINT_USING "#####",32*x+y; : PRINT " "; : STRIP
       32*x+y:PRINT "     "
190   NEXT y
200  NEXT x
210  PAUSE
```

comment: This program requires QPC to be operating in at least an 800x600 pixel screen mode.

## COLOUR_PAL

This is a 256 colour mode, which allows you to display any 256 colours from a palette of 16 Million. Colour specification in the **COLOUR_PAL** mode, is defined as a number between 0 and 255

example: i. **PAPER 63 : CLS** Deep purple
ii. **PAPER 35 : CLS** Pastel Yellow

This table lists all the standard 256 colours available in **COLOUR_PAL**, along with their 24 Bit values.

| Colour Number | | Colour Name | 24 Bit value R | G | B | Colour Number | | Colour Name | 24 Bit value R | G | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | Black | 00 | 00 | 00 | 13 | 0D | Ash grey | DB | DB | DB |
| 1 | 01 | White | FF | FF | FF | 14 | 0E | Dark red | 92 | 00 | 00 |
| 2 | 02 | Red | FF | 00 | 00 | 15 | 0F | Light green | B6 | FF | B6 |
| 3 | 03 | Green | 00 | FF | 00 | 16 | 10 | Mustard | 92 | 92 | 00 |
| 4 | 04 | Blue | 00 | 00 | FF | 17 | 11 | Dark green | 00 | 92 | 00 |
| 5 | 05 | Magenta | FF | 00 | FF | 18 | 12 | Sea blue | 00 | 92 | 92 |
| 6 | 06 | Yellow | FF | FF | 00 | 19 | 13 | Dark blue | 00 | 00 | 92 |
| 7 | 07 | Cyan | 00 | FF | FF | 20 | 14 | Purple | 92 | 00 | 92 |
| 8 | 08 | Dark slate | 24 | 24 | 24 | 21 | 15 | Shocking pink | FF | 00 | 92 |
| 9 | 09 | Slate grey | 49 | 49 | 49 | 22 | 16 | Orange | FF | 92 | 00 |
| 10 | 0A | Dark grey | 6D | 6D | 6D | 23 | 17 | Lime green | 92 | FF | 00 |
| 11 | 0B | Grey | 92 | 92 | 92 | 24 | 18 | Apple green | 00 | FF | 92 |
| 12 | 0C | Light grey | B6 | B6 | B6 | 25 | 19 | Bright blue | 00 | 92 | FF |

| Colour Number | | Colour Name | 24 Bit value R | G | B |
|---|---|---|---|---|---|
| 26 | 1A | Mauve | 92 | 00 | FF |
| 27 | 1B | Peach | FF | B6 | B6 |
| 28 | 1C | Light yellow | FF | FF | B6 |
| 29 | 1D | Light blue | B6 | FF | FF |
| 30 | 1E | Sky blue | B6 | B6 | FF |
| 31 | 1F | Rose pink | FF | B6 | FF |
| 32 | 20 | Pink | FF | B6 | DB |
| 33 | 21 | Beige | FF | DB | B6 |
| 34 | 22 | Pastel pink | FF | DB | DB |
| 35 | 23 | Pastel yellow | FF | FF | DB |
| 36 | 24 | Pastel green | DB | FF | DB |
| 37 | 25 | Pastel cyan | DB | FF | FF |
| 38 | 26 | Pastel blue | DB | DB | FF |
| 39 | 27 | Pastel rose | FF | DB | FF |
| 40 | 28 | Brick | B6 | 6D | 6D |
| 41 | 29 | Light khaki | B6 | B6 | 6D |
| 42 | 2A | Dull green | 6D | B6 | 6D |
| 43 | 2B | Dull cyan | 6D | B6 | B6 |
| 44 | 2C | Steel blue | 6D | 6D | B6 |
| 45 | 2D | Dull pink | B6 | 6D | B6 |
| 46 | 2E | Brown | 6D | 24 | 24 |
| 47 | 2F | Khaki | 6D | 6D | 24 |
| 48 | 30 | Dusky green | 24 | 6D | 24 |
| 49 | 31 | Dusky blue | 24 | 6D | 6D |
| 50 | 32 | Midnight blue | 24 | 24 | 6D |
| 51 | 33 | Plum | 6D | 24 | 6D |
| 52 | 34 | Dusky pink | B6 | 49 | 92 |
| 53 | 35 | Buff | B6 | 92 | 49 |
| 54 | 36 | Avocado | 92 | B6 | 49 |
| 55 | 37 | Dull turquoise | 49 | B6 | 92 |
| 56 | 38 | Dull blue | 49 | 92 | B6 |
| 57 | 39 | Faded purple | 92 | 49 | B6 |
| 58 | 3A | Cerise | 92 | 00 | 49 |
| 59 | 3B | Tan | 92 | 49 | 00 |
| 60 | 3C | Grass green | 49 | 92 | 00 |
| 61 | 3D | Sea green | 00 | 92 | 49 |
| 62 | 3E | Ultramarine | 00 | 49 | 92 |
| 63 | 3F | Deep purple | 49 | 00 | 92 |
| 64 | 40 | | 00 | 00 | 49 |
| 65 | 41 | | 24 | 00 | 24 |
| 66 | 42 | | 24 | 00 | 6D |
| 67 | 43 | | 24 | 00 | B6 |
| 68 | 44 | | 24 | 00 | FF |
| 69 | 45 | | 49 | 00 | 00 |
| 70 | 46 | | 49 | 00 | 49 |
| 71 | 47 | | 49 | 00 | DB |
| 72 | 48 | | 6D | 00 | 24 |
| 73 | 49 | | 6D | 00 | 6D |
| 74 | 4A | | 6D | 00 | B6 |
| 75 | 4B | | 6D | 00 | FF |
| 76 | 4C | | B6 | 00 | 24 |
| 77 | 4D | | B6 | 00 | 6D |
| 78 | 4E | | B6 | 00 | B6 |
| 79 | 4F | | B6 | 00 | FF |
| 80 | 50 | | DB | 00 | 00 |
| 81 | 51 | | DB | 00 | 49 |
| 82 | 52 | | DB | 00 | 92 |
| 83 | 53 | | DB | 00 | DB |
| 84 | 54 | | FF | 00 | 6D |
| 85 | 55 | | 00 | 24 | 00 |
| 86 | 56 | | 00 | 24 | 49 |
| 87 | 57 | | 00 | 24 | 92 |
| 88 | 58 | | 00 | 24 | DB |
| 89 | 59 | | 24 | 24 | B6 |
| 90 | 5A | | 24 | 24 | FF |
| 91 | 5B | | 49 | 24 | 00 |
| 92 | 5C | | 49 | 24 | 49 |
| 93 | 5D | | 49 | 24 | 92 |
| 94 | 5E | | 49 | 24 | DB |

| Colour Number | | Colour Name | 24 Bit value R | G | B |
|---|---|---|---|---|---|
| 95 | 5F | | 6D | 24 | B6 |
| 96 | 60 | | 6D | 24 | FF |
| 97 | 61 | | 92 | 24 | 00 |
| 98 | 62 | | 92 | 24 | 49 |
| 99 | 63 | | 92 | 24 | 92 |
| 100 | 64 | | 92 | 24 | DB |
| 101 | 65 | | B6 | 24 | 24 |
| 102 | 66 | | B6 | 24 | 6D |
| 103 | 67 | | B6 | 24 | B6 |
| 104 | 68 | | B6 | 24 | FF |
| 105 | 69 | | DB | 24 | 00 |
| 106 | 6A | | DB | 24 | 49 |
| 107 | 6B | | DB | 24 | 92 |
| 108 | 6C | | DB | 24 | DB |
| 109 | 6D | | FF | 24 | 24 |
| 110 | 6E | | FF | 24 | 6D |
| 111 | 6F | | FF | 24 | B6 |
| 112 | 70 | | FF | 24 | FF |
| 113 | 71 | | 00 | 49 | 00 |
| 114 | 72 | | 00 | 49 | 49 |
| 115 | 73 | | 00 | 49 | DB |
| 116 | 74 | | 24 | 49 | 24 |
| 117 | 75 | | 24 | 49 | 6D |
| 118 | 76 | | 24 | 49 | B6 |
| 119 | 77 | | 24 | 49 | FF |
| 120 | 78 | | 49 | 49 | 00 |
| 121 | 79 | | 49 | 49 | 92 |
| 122 | 7A | | 49 | 49 | DB |
| 123 | 7B | | 6D | 49 | 24 |
| 124 | 7C | | 6D | 49 | 6D |
| 125 | 7D | | 6D | 49 | B6 |
| 126 | 7E | | 6D | 49 | FF |
| 127 | 7F | | 92 | 49 | 49 |
| 128 | 80 | | 92 | 49 | DB |
| 129 | 81 | | B6 | 49 | 24 |
| 130 | 82 | | B6 | 49 | 6D |
| 131 | 83 | | B6 | 49 | FF |
| 132 | 84 | | DB | 49 | 00 |
| 133 | 85 | | DB | 49 | 49 |
| 134 | 86 | | DB | 49 | 92 |
| 135 | 87 | | DB | 49 | DB |
| 136 | 88 | | FF | 49 | 24 |
| 137 | 89 | | FF | 49 | 6D |
| 138 | 8A | | FF | 49 | B6 |
| 139 | 8B | | FF | 49 | FF |
| 140 | 8C | | 00 | 6D | 00 |
| 141 | 8D | | 00 | 6D | 49 |
| 142 | 8E | | 00 | 6D | 92 |
| 143 | 8F | | 00 | 6D | DB |
| 144 | 90 | | 24 | 6D | B6 |
| 145 | 91 | | 24 | 6D | FF |
| 146 | 92 | | 49 | 6D | 00 |
| 147 | 93 | | 49 | 6D | 49 |
| 148 | 94 | | 49 | 6D | 92 |
| 149 | 95 | | 49 | 6D | DB |
| 150 | 96 | | 6D | 6D | FF |
| 151 | 97 | | 92 | 6D | 00 |
| 152 | 98 | | 92 | 6D | 49 |
| 153 | 99 | | 92 | 6D | 92 |
| 154 | 9A | | 92 | 6D | DB |
| 155 | 9B | | B6 | 6D | 24 |
| 156 | 9C | | B6 | 6D | FF |
| 157 | 9D | | DB | 6D | 00 |
| 158 | 9E | | DB | 6D | 49 |
| 159 | 9F | | DB | 6D | 92 |
| 160 | A0 | | DB | 6D | DB |
| 161 | A1 | | FF | 6D | 24 |
| 162 | A2 | | FF | 6D | 6D |
| 163 | A3 | | FF | 6D | B6 |

| Colour Number | Colour Name | 24 Bit value R | G | B |
|---|---|---|---|---|
| 164 | A4 | FF | 6D | FF |
| 165 | A5 | 24 | 92 | 24 |
| 166 | A6 | 24 | 92 | 6D |
| 167 | A7 | 24 | 92 | B6 |
| 168 | A8 | 24 | 92 | FF |
| 169 | A9 | 49 | 92 | 49 |
| 170 | AA | 49 | 92 | DB |
| 171 | AB | 6D | 92 | 24 |
| 172 | AC | 6D | 92 | 6D |
| 173 | AD | 6D | 92 | B6 |
| 174 | AE | 6D | 92 | FF |
| 175 | AF | 92 | 92 | 49 |
| 176 | B0 | 92 | 92 | DB |
| 177 | B1 | B6 | 92 | 24 |
| 178 | B2 | B6 | 92 | B6 |
| 179 | B3 | B6 | 92 | FF |
| 180 | B4 | DB | 92 | 00 |
| 181 | B5 | DB | 92 | 49 |
| 182 | B6 | DB | 92 | 92 |
| 183 | B7 | DB | 92 | DB |
| 184 | B8 | FF | 92 | 6D |
| 185 | B9 | FF | 92 | B6 |
| 186 | BA | FF | 92 | FF |
| 187 | BB | 00 | B6 | 00 |
| 188 | BC | 00 | B6 | 49 |
| 189 | BD | 00 | B6 | 92 |
| 190 | BE | 00 | B6 | DB |
| 191 | BF | 24 | B6 | 24 |
| 192 | C0 | 24 | B6 | 6D |
| 193 | C1 | 24 | B6 | B6 |
| 194 | C2 | 24 | B6 | FF |
| 195 | C3 | 49 | B6 | 00 |
| 196 | C4 | 49 | B6 | 49 |
| 197 | C5 | 49 | B6 | DB |
| 198 | C6 | 6D | B6 | 24 |
| 199 | C7 | 6D | B6 | FF |
| 200 | C8 | 92 | B6 | 00 |
| 201 | C9 | 92 | B6 | 92 |
| 202 | CA | 92 | B6 | DB |
| 203 | CB | B6 | B6 | 24 |
| 204 | CC | DB | B6 | 00 |
| 205 | CD | DB | B6 | 49 |
| 206 | CE | DB | B6 | 92 |
| 207 | CF | DB | B6 | DB |
| 208 | D0 | FF | B6 | 24 |
| 209 | D1 | FF | B6 | 6D |

| Colour Number | Colour Name | 24 Bit value R | G | B |
|---|---|---|---|---|
| 210 | D2 | 00 | DB | 00 |
| 211 | D3 | 00 | DB | 49 |
| 212 | D4 | 00 | DB | 92 |
| 213 | D5 | 00 | DB | DB |
| 214 | D6 | 24 | DB | 24 |
| 215 | D7 | 24 | DB | 6D |
| 216 | D8 | 24 | DB | B6 |
| 217 | D9 | 24 | DB | FF |
| 218 | DA | 49 | DB | 00 |
| 219 | DB | 49 | DB | 49 |
| 220 | DC | 49 | DB | 92 |
| 221 | DD | 49 | DB | DB |
| 222 | DE | 6D | DB | 24 |
| 223 | DF | 6D | DB | 6D |
| 224 | E0 | 6D | DB | B6 |
| 225 | E1 | 6D | DB | FF |
| 226 | E2 | 92 | DB | 00 |
| 227 | E3 | 92 | DB | 49 |
| 228 | E4 | 92 | DB | 92 |
| 229 | E5 | 92 | DB | DB |
| 230 | E6 | B6 | DB | 24 |
| 231 | E7 | B6 | DB | 6D |
| 232 | E8 | B6 | DB | B6 |
| 233 | E9 | B6 | DB | FF |
| 234 | EA | DB | DB | 49 |
| 235 | EB | DB | DB | 92 |
| 236 | EC | FF | DB | 6D |
| 237 | ED | 00 | FF | 49 |
| 238 | EE | 24 | FF | 6D |
| 239 | EF | 24 | FF | B6 |
| 240 | F0 | 49 | FF | 00 |
| 241 | F1 | 49 | FF | 49 |
| 242 | F2 | 49 | FF | 92 |
| 243 | F3 | 49 | FF | DB |
| 244 | F4 | 6D | FF | 24 |
| 245 | F5 | 6D | FF | 6D |
| 246 | F6 | 6D | FF | B6 |
| 247 | F7 | 6D | FF | FF |
| 248 | F8 | 92 | FF | 49 |
| 249 | F9 | 92 | FF | 92 |
| 250 | FA | 92 | FF | DB |
| 251 | FB | B6 | FF | 24 |
| 252 | FC | B6 | FF | 6D |
| 253 | FD | DB | FF | 49 |
| 254 | FE | DB | FF | 92 |
| 255 | FF | FF | FF | 6D |

This program will display all of the colours available in the **COLOUR_PAL** mode.

```
100  REMark COLOUR_PAL colours
110  WINDOW 750,550,25,25
120  COLOUR_PAL
130  PAPER 0:INK 1
140  CLS
150  FOR x=0 TO 7
160  FOR y= 0 TO 31
170   PAPER 0:INK 1
180   AT y,15*x : PRINT_USING "#####",32*x+y; : PRINT " "; : STRIP
      (32*x+y):PRINT "     "
190   NEXT y
200  NEXT x
210  PAUSE
```

comment: This program requires QPC to be operating in at least an 800x600 pixel screen mode.

## COLOUR_24

This is a 16 Million (24 Bit) colour mode, allowing you to display any of the available 16 Million colours.

Colour specification in the COLOUR_24 mode, is defined as a number between 0 and 16,777,215


The 24 Bit value used in **INK**, **PAPER**, **STRIP** etc is calculated as (Red * 65536) + (Green * 256) + Blue. Where each of the colours Red, Green, and Blue have values between 0 and 255.

| | | | |
|---|---|---|---|
| example: | i. | **PAPER 219 * 65536 + 219 * 256 + 255 : CLS** | Pastel Blue |
| | ii. | **PAPER 14408703 : CLS** | as above in decimal |
| | ii. | **PAPER $DBDBFF : CLS** | as above in hexadecimal |

## COLOUR_NATIVE

The format accepted by **COLOUR_NATIVE** depends on the colour mode QPC is currently running in. It uses the same values as the pixels on the screen.

## Palettes

The actual colours used in **COLOUR_QL** and **COLOUR_PAL** mode are stored in tables which may be redefined, allowing you to change the colours that are displayed to any of the available 16 Million (24 Bit) colours.

**PALETTE_QL** enables you to redefine the 8 colours used in the **COLOUR_QL** mode, to any of the 16 Million (24 Bit) colours. The replacement colours must be spacified as 24 Bit true colours.

**PALETTE_QL** *start*, *colour* [,*colour* [,*colour* [,*colour* [,*colour* [,*colour* [,*colour* ]]]]]]

*start* := *first colour in table to change*
*colour* := *true colour value*

This program will change only the two colours cyan and yellow, into khaki and orange, leaving the others unchanged.

```
600  khaki = 109*65536+109*256+36
610  orange = 255*65536+146*256
620  PALETTE_QL 5,khaki,orange : REMark change only 2 colours
```

Many QL programs define some of the colours displayed as "white minus a colour", on a 4 colour QL display, "white minus red" appears as green on a QL . "white minus red" however is really cyan, not green. As a result, many QL mode 4 programs take on rainbow hues when displayed on a QPC screen.

This can be "fixed" by redefining the colours so that colour 2 (Red) is a bright crimson, and colour 4 (Green) is a bright sea green. This will ensure that using true colours (24 Bit), colour 2 plus colour 4 is equal to colour 7.
We also need to ensure that colour 1 is equal to colour 0, colour 3 is equal to colour 2, colour 5 is equal to colour 4, and colour 6 is equal to colour 7. This is to simulate the QL mode 4 colours.

```
600  crimson = 255*65536+100 : REMark crimson is red + a bit of blue
610  sea = 255*256+155 : REMark sea green is green + the rest of blue
620  white = crimson + sea
630  PALETTE_QL 0,0,0,crimson,crimson,sea,sea,white,white :
     REMark set 8 colours
```

**PALETTE_8** enables you to redefine any or all of the 256 colours used in the **COLOUR_PAL** mode to any of the 16 Million (24 Bit) colours. The replacement colours must be specified as a 24 Bit true colour.

**PALETTE_8** *start*, *colour* * [,*colour* ] *

*start* := *first colour in table to change*
*colour* := *true colour value*

If new colours are required, they should replace colours towards the top of the table so that the low colours remain unchanged.

This example will set colours 248 to 255 of PALETTE_8 to black, blue, red, magenta, green, cyan, yellow, and white

```
100  black = 0 : red = 255 * 65536 : green = 255 * 256 : blue = 255
110  magenta = red + blue : cyan = blue + green : yellow = green + red
120  white = red + green + blue
130  PALETTE_8 248, black, blue, red, magenta, green, cyan, yellow, white
```

**warning:** Once a palette has been changed it can only be reset manually or by resetting SMSQ/E.

| Command | Function |
|---|---|
| **COLOUR_QL** | set 8 colour mode |
| **COLOUR_PAL** | set 256 colour mode |
| **COLOUR_24** | set 16 Million (24 Bit) mode |
| **PALETTE_QL** | change 8 colour palette |
| **PALETTE_8** | change 256 colour palette |

# window manager colour palettes

The Windows Manager maintains a set of standard colour schemes that can be used to provide a consistent appearance of Windows on the screen.

The commands **WM_PAPER**, **WM_STRIP**, **WM_INK**, **WM_BORDER**, and **WM_BLOCK** perform much the same functions as **PAPER**, **STRIP**, **INK**, **BORDER**, and **BLOCK**. But use one of the seven Window Manager colour schemes, defined as a 16 Bit word (a number in the range 0 to 65535).

## Simple colour palette scheme

This colour scheme corresponds to the **COLOUR_QL** colour mode. The first byte of the colour word has a value of zero, and the second byte, a value in the range 0 to 255 to represent the solid, or stipple colour required.

example: i. **WM_PAPER $0002 : CLS** {red}
ii. **WM_PAPER $009F : CLS** {green and white vertical stripes}

## The colour palette scheme

This colour scheme corresponds to the **COLOUR_PAL** colour mode. The first byte of the colour word has a value of one, and the second byte, a value in the range 0 to 255 to represent the colour required.

example: i. **WM_PAPER $0112 : CLS** {sea blue}
ii. **WM_PAPER $013A : CLS** {cerise}

## The system palette schemes

This colour scheme corresponds to the colour modes used in Pointer Environment programs. The first byte of the colour word has a value of two, and the second byte, a value in the range 0 to the value of **SP_GETCOUNT** minus 1, to represent the colour required.

The system palette colour scheme is further divided into four sub colour schemes, which are selected by using the **SP_JOBPAL** command. They default to the following colour schemes but can be changed at runtime:

0 White paper, with black ink. With a green and white striped title bar.
1 Black paper, with white ink. With a red and black striped title bar.
2 White paper, with black ink. With a red and white striped title bar.
3 Black paper, with white ink. With a green and black striped title bar.

Each element of a Pointer Environment window has a colour, (or stipple pattern) which is associated with it as defined in the table below.

To provide consistency in Pointer Environment programs, you should use the appropriate colours in the table below (although you do not have to).

| Number | Window element |
|---|---|
| $0200 | Window border |
| $0201 | Window background |
| $0202 | Window foreground |
| $0203 | Window middleground |
| $0204 | Title background |
| $0205 | Title text background |
| $0206 | Title foreground |
| $0207 | Loose item highlight |
| $0208 | Loose item available background |

| Number | Window element |
|---|---|
| $0209 | Loose item available foreground |
| $020a | Loose item selected background |
| $020b | Loose item selected foreground |
| $020c | Loose item unavailable background |
| $020d | Loose item unavailable foreground |
| $020e | Information window border |
| $020f | Information window background |
| $0210 | Information window foreground |
| $0211 | Information window middleground |
| $0212 | Subsidiary information window border |
| $0213 | Subsidiary information window background |
| $0214 | Subsidiary information window foreground |
| $0215 | Subsidiary information window middleground |
| $0216 | Application window border |
| $0217 | Application window background |
| $0218 | Application window foreground |
| $0219 | Application window middleground |
| $021a | Application window item highlight |
| $021b | Application window item available background |
| $021c | Application window item available foreground |
| $021d | Application window item selected background |
| $021e | Application window item selected foreground |
| $021f | Application window item unavailable background |
| $0220 | Application window item unavailable foreground |
| $0221 | Pan/scroll bar |
| $0222 | Pan/scroll bar section |
| $0223 | Pan/scroll bar arrow |
| $0224 | Button highlight |
| $0225 | Button border |
| $0226 | Button background |
| $0227 | Button foreground |
| $0228 | Hint border |
| $0229 | Hint background |
| $022a | Hint foreground |
| $022b | Hint middleground |
| $022c | Error message background |
| $022d | Error message foreground |
| $022e | Error message middleground |
| $022f | Shaded area |
| $0230 | Dark 3D border shade |
| $0231 | Light 3D border shade |
| $0232 | Vertical area fill |
| $0233 | Subtitle background |
| $0234 | Subtitle text background |
| $0235 | Subtitle foreground |
| $0236 | Menu index background |
| $0237 | Menu index foreground |
| $0238 | Separator lines etc. |

example: The following program will display a message on the screen, in the title colours of the System palette scheme number 2. That is black text with a red and white striped title bar.

```
10 SP_JOBPAL –1,2                {select system palette scheme 2 for this job}
20 WM_PAPER $0204 : CLS 3        {set title background colour}
30 WM_STRIP $0205                {set title text background colour}
40 WM_INK $0206                  {set title text colour}
50 AT 0,10 : PRINT ;"  Title bar colours  "
```

## Grey scale palette scheme

This colour scheme provides a series of shades of grey. The first byte of the colour word has a value of three, and the second byte, a value in the range 0 to 255 to represent the shade of grey required.

example:
i. **WM_PAPER $0300 : CLS** {black}
ii. **WM_PAPER $0380 : CLS** {mid grey}
ii. **WM_PAPER $03FF : CLS** {white}

## Border colours palette scheme

This colour scheme, provides a combination of border styles and colours. The actual colours used in this palette scheme, depend on which system palette colour scheme has been selected by the **SP_JOBPAL** command. The first byte of the colour word has a value of four, and the second byte, a value in the range 0 to 15 to represent one of the eight border styles.

| Number | Border style |
|---|---|
| $0400 | 3D Chiselled Border (Button Raised) |
| $0401 | As above with colours swapped |
| $0402 | 3D Chiselled Border (Button Depressed) |
| $0403 | As above with colours swapped |
| $0404 | 3D Raised Border |
| $0405 | As above with colours swapped |
| $0406 | 3D Chiselled Border with shadow on south east side |
| $0407 | 3D Chiselled Border with shadow on north west side |
| $0408 | Border on left hand side only |
| $0409 | As above with colours swapped |
| $040A | Border on right hand side only |
| $040B | As above with colours swapped |
| $040C | Border on top side only |
| $040D | As above with colours swapped |
| $040E | Border on bottom side only |
| $040F | As above with colours swapped |

example: The following program will display all of the border styles.

```
100  PAPER 4 : CLS
110  JOB_PAL –1, 0
120  WINDOW 200, 100, 100, 100
130  FOR x = 0 TO 15
140   WM_BORDER 4, $400 + x
150   AT 0, 0 : Print "Border Style " ; x
160   PAUSE
170   WINDOW 200, 100, 100, 100
180  END FOR x
```

Some of those borders styles have widths that are not compatible with the traditional QL borders which can cause compatibility problems with applications not prepared for this. Therefore so called "compatiblity modes" are available, too. When a compatiblity mode is selected the border has the same width as a traditional QL border with the additional space

filled differently depending on the mode. There are 3 different compatibility modes available which can be specified in the upper halve of the low byte, i.e. $04x0.

| Number | Border style |
|---|---|
| $040x | No compatiblity mode |
| $041x | Additional space is outside of border and left untouched |
| $042x | Additional space is inside of border and filled with the paper colour |
| $043x | Additional space is outside of border and filled with the paper colour |

## Palette stipples scheme

This colour scheme is produced as a combination of two colours combined in a stipple pattern. The most significant two bits of the first byte have a binary value of %01. The next two binary bits contain the stipple type. The next 4 binary bits, and the top two binary bits of the second byte contain the stipple colour, and the last six binary bits contain the main colour.

The values used for the main and the stipple colours are taken from the first 64 colours of the **COLOUR_PAL**, 256 colour mode.

example: The following program will produce a white background with a khaki stipple pattern 0

```
100  kahaki = $2F
110  white = $01
120  WM_PAPER $4000 + (64 * khaki) + white
130  CLS
```

## 15 Bit RGB scheme

This colour scheme provides 32 thousand colour combinations. The first binary bit of the colour word has a binary value of 1. The next five binary bits represent the red component of the colour. The next five binary bits represent the green component of the colour, and the last five binary bits represent the blue component of the colour.

example: The following program will create a Magenta background

```
100  red = 31                    {range 0..31}
110  green = 0
120  blue = 31
130  WM_PAPER $8000 + (1024 * red) + (32 * green) + blue
140  CLS
```

# communications
# parallel

QPC can access up to 8 parallel ports (called PAR1, PAR2, etc) for connecting it to equipment which use parallel output communications.

The PC on which you are running QPC will usually have one parallel port fitted, known as LPT1. The QPC Configurator can determine which PAR port is connected to which LPT port. (usually PAR1 = LPT1 and PAR2 = LPT2). A parallel port can also be connected to the spool job of a printer (by configuring the name of the desired printer), thus enabling the access to USB and network printers.

The PC parallel port connectors will usually be 25 pin connectors

Translate, determines whether the data sent should be translated into other characters. This is generally used when sending text to printers, to convert the ASCII codes which are different between the QPC character set, and the printers characters set. See the **TRA** command.

Parallel communications on QPC are 'simplex', that is the parallel port is transmit only.

# communications
# serial RS-232-C

QPC can access up to 8 serial ports (called SER1, SER2, etc) for connecting it to equipment which use serial communications obeying EIA standard RS-232-C or a compatible standard.

The RS-232-C 'standard' was originally designed to enable computers to send and receive data via telephone lines using a modem. However, it is now frequently used to connect computers directly with each other and to various items of peripheral equipment, e.g. printers, modems, etc.

As the RS-232-C 'standard' manifests itself in many different forms on different pieces of equipment, it can be an extremely difficult job, even for an expert to connect together for the first time two pieces of supposedly standard RS-232-C equipment. This section will attempt to cover most of the basic problems that you may encounter.

The PC on which you are running QPC will usually have one or two serial ports fitted, known as COM1 and COM2. The QPC Configurator can determine which SER port is connected to which COM port. (usually SER1 = COM1 and SER2 = COM2)

The PC serial port connectors will be either 9, or 25 pin connectors

| 9 pin | 25 pin | Name | Function | Direction |
|---|---|---|---|---|
| 1 | 8 | DCD | Data Carrier Detect | In |
| 2 | 3 | RXD | Receive Data | In |
| 3 | 2 | TXD | Transmit Data | Out |
| 4 | 20 | DTR | Data Terminal Ready | Out |
| 5 | 7 | GND | Signal Ground | - |
| 6 | 6 | DSR | Data Set Ready | In |
| 7 | 4 | RTS | Ready to Send | Out |
| 8 | 5 | CTR | Clear To Send | In |
| 9 | 22 | RI | Ring Indicator | In |

Once the equipment has been connected, the baud rate (the speed of transmission of data) must be set so that the baud rates for both QPC and the connected equipment are the same. The serial ports on QPC can be set to operate at:

| | |
|---|---|
| 300 | |
| 600 | |
| 1200 | |
| 2400 | |
| 4800 | |
| 9600 | |
| 19200 | |
| 38400 | |
| 57600 | |
| 115200 | baud |

The QPC baud rate for each serial port is set by the **BAUD** command.

The parity to be used by QPC must also be set to match that expected by the peripheral equipment. Parity is usually used to detect simple transmission errors and may be set to be even, odd, mark, space or no parity, i.e. all 8 bits of the byte are used for data.

Flow control determines how QPC and the peripheral device know when to communicate with each other. Flow control can be either:

| | |
|---|---|
| Hardware | Where a signal line is used by one end of the connection to the other end, to say, don't talk now I'm busy. |
| Software | Where a signal is sent down the Transmit data line to the receiver, to say, don't talk now I'm busy (XOFF), or I am now ready to listen (XON). The receiver can be either the peripheral device, or QPC itself |
| None | There is no flow control. Data will be lost, or corrupted if the receiver is busy doing other things when data arrives, or cannot process the data it is receiving fast enough. |

Translate, determines whether the data sent should be translated into other characters. This is generally used when sending text to printers, to convert the ASCII codes which are different between the QPC character set, and the printers characters set. See the **TRA** command.

Serial communications on QPC are 'full duplex', that is both receive and transmit can operate concurrently.

The parity and handshaking are selected when the serial channel is opened.

comment: There is also the serial receive only device (SRX), and serial transmit only device (STX). They are the same as the SER device, except that one will only transmit data, and the other will only receive data.

| command | function |
|---|---|
| **BAUD** | set transmission speed |
| **OPEN** | open serial channels * |
| **CLOSE** | close serial channels |

* see concept *device* for a full specification

## cursor sprites

The **CURSPRLOAD** command may be used to replace the standard red block cursor with a user-defined replacement. This replacement cursor must be the same size (6 by 10 pixels) as the standard cursor, but may be any colour or pattern the user requires.

These replacement cursors may only be used in windows that have a standard character size of 6 by 10 pixels (**CSIZE 0,0**).

A cursor sprite definition comprises of two parts, The sprite header and the sprite data.

The sprite header is defined as follows:

| Offset | Size | Description |
|---|---|---|
| $00 | byte | sprite mode |
| $01 | byte | colour mode/system sprite number |
| $02 | byte | dynamic sprite version number |
| $03 | byte | sprite control |
| $04 | word | X size |
| $06 | word | Y size |
| $08 | word | X offset |
| $0A | word | Y offset |
| $0C | long | relative pointer to colour pattern |
| $10 | long | relative pointer to mask/alpha channel |
| $14 | long | relative pointer to next object |
| $18 | long | relative pointer to options |
| $1C | long | relative pointer to sprite block |

This section is just an introduction to creating sprites and further information on the construction of sprites may be found elsewhere.

The following two example programs will create 256 colour, **COLOUR_PAL** mode, sprite definition files that may be used with the **CURSPRLOAD** command.

The following example program will produce a cursor sprite of a white arrow in a red block.

```
100 OPEN_OVER#3,flp1_arrow_spr      {create a file for our sprite definition}
110 RESTORE
120 REPeat loop
130  IF EOF() THEN EXIT loop
140  READ x:BPUT#3,x                {read data and store in file}
150 END REPeat loop
160 CLOSE#3
170 CURSPRLOAD flp1_arrow_spr       {load our new cursor sprite}
180 CURSPRON  –1                    {activate it for this job}
1000 DATA 2                         {start of header, GD2 sprite mode}
1010 DATA 31                        {8 bit palette mapped colour mode}
1020 DATA 0                         {leave version number as 0}
1030 DATA 32                        {alpha channel is present}
1040 DATA 0,6                       {cursor sprite size is always 6 x 10}
1050 DATA 0,10
1060 DATA 0,3                       {cursor sprit offset}
1070 DATA 0,4
1080 DATA 0,0,0,20                  {pointer to pattern}
1090 DATA 0,0,0,96                  {pointer to alpha channel}
1100 DATA 0,0,0,0                   {leave as 0}
1110 DATA 0,0,0,0                   {leave as 0}
1120 DATA 0,0,0,0                   {leave as 0}
1130 REMark Sprite
1140 DATA 2,2,2,2,2,2,0,0           {pattern data for sprite}
1150 DATA 2,2,2,2,1,2,0,0           {in palette mode 2 is red, & 1 is white}
1160 DATA 2,2,2,2,1,2,0,0           {note the 0's padding the line out}
1170 DATA 2,2,2,2,1,2,0,0           {to 8 bytes}
1180 DATA 2,2,2,2,1,2,0,0
1190 DATA 2,2,1,2,1,2,0,0
1200 DATA 2,1,2,2,1,2,0,0
1210 DATA 1,1,1,1,1,2,0,0
1220 DATA 2,1,2,2,2,2,0,0
1230 DATA 2,2,1,2,2,2,0,0
1240 REMark alpha channel
1250 DATA 255,255,255,255,255,255   {alpha channel data}
1260 DATA 255,255,255,255,255,255   {note no padding of the line}
1270 DATA 255,255,255,255,255,255
1280 DATA 255,255,255,255,255,255
1290 DATA 255,255,255,255,255,255
1300 DATA 255,255,255,255,255,255
1310 DATA 255,255,255,255,255,255
1320 DATA 255,255,255,255,255,255
1330 DATA 255,255,255,255,255,255
1340 DATA 255,255,255,255,255,255
```

The alpha channel allows a gradual mix between the background and the sprite pattern. Every pixel of the sprite is represented by one byte of the alpha channel. 0 means that the pixel of the sprite is completely transparent, and 255 means that the pixel of the sprite is completely opaque. Values in between determine the amount of mixing of the background and foreground.

This second example will create a green underscore cursor sprite.

```
100 OPEN_OVER#3,flp1_under_spr
110 RESTORE
120 REPeat loop
130  IF EOF() THEN EXIT loop
140  READ x:BPUT#3,x
150 END REPeat loop
160 CLOSE#3
170 CURSPRLOAD flp1_under_spr
180 CURSPRON -1
1000 DATA 2                          {header information as before}
1010 DATA 31
1020 DATA 0
1030 DATA 32
1040 DATA 0,6
1050 DATA 0,10
1060 DATA 0,3
1070 DATA 0,4
1080 DATA 0,0,0,20
1090 DATA 0,0,0,96
1100 DATA 0,0,0,0
1110 DATA 0,0,0,0
1120 DATA 0,0,0,0
1130 REMark Sprite
1140 DATA 4,4,4,4,4,4,0,0            {pattern data for sprite}
1150 DATA 4,4,4,4,4,4,0,0            {3 is green, & 4 is blue}
1160 DATA 4,4,4,4,4,4,0,0            {the blue could be any colour}
1170 DATA 4,4,4,4,4,4,0,0            {as it is never seen due to the}
1180 DATA 4,4,4,4,4,4,0,0            {alpha channel}
1190 DATA 4,4,4,4,4,4,0,0
1200 DATA 4,4,4,4,4,4,0,0
1210 DATA 4,4,4,4,4,4,0,0
1220 DATA 4,4,4,4,4,4,0,0
1230 DATA 3,3,3,3,3,3,0,0
1240 REMark alpha channel
1250 DATA 0,0,0,0,0,0                {all pixels of the sprite are}
1260 DATA 0,0,0,0,0,0                {transparent except the last line}
1270 DATA 0,0,0,0,0,0
1280 DATA 0,0,0,0,0,0
1290 DATA 0,0,0,0,0,0
1300 DATA 0,0,0,0,0,0
1310 DATA 0,0,0,0,0,0
1320 DATA 0,0,0,0,0,0
1330 DATA 0,0,0,0,0,0
1340 DATA 255,255,255,255,255,255
```

# data types
# variables

**integer** Integers are whole numbers in the range -32768 to +32767. Variables are assumed to be integer if the variable identifier is suffixed with a percent %. There are no integer constants in SBASIC, so all constants are stored as floating point numbers.

syntax: *identifier*%

example: i. **counter%**
ii. **size_limit%**
iii. **this_is_an_integer_variable%**

**floating point**
Floating point numbers are in the range +/- (10^-615 to 10^615), with 8 significant digits. Floating point is the default data type in SBASIC. All constants are held in floating point form and can be entered using exponent notation.

syntax: *identifier* | *constant*

example: i. **current accumulation**
ii. **76.2356**
iii. **354E25**

**string** A string is a sequence of characters up to 32766 characters long. Variables are assumed to be type string if the variable name is suffixed by a $. String data is represented by enclosing the required characters in either single or double quotation marks.

syntax: *identifier*$ | "*text*"

example: i. **string_variables$**
ii. **"this is string data"**
iii. **"this is another string"**

**name** Type name has the same form as a standard SBASIC identifier and is used by the system to name Floppy disk files etc.

syntax: *identifier*

example: i. **flp1_data_file**
ii. **ser1e**

**binary** Binary values are represented as a sequence of zeros and ones, preceded by a percentage sign.

syntax: **%***constant*

example: i. **%1001**
ii. **%11001010**

**hexadecimal**
Hexadecimal values are represented by a sequence of the numbers 0 – 9 and the letters A – F (to represent the values 0 – 15), preceded by a dollar sign.

syntax: **$***constant*

example: i. **Ten = $A**
ii. **one_hundred = $64**
iii. **PRINT PEEK($28000)**

# dev
# virtual device
DEV is a defaulting device that provides up to 8 default search paths to be used when opening files. As it was designed to be dumped on top of QDOS it is not very clean, but, equally, it is reasonably efficient.

Each DEV (DEV1 to DEV8) device is attached to a particular real device or a particular default directory on a real device.

Files on a DEV device can be **OPEN**ed, used and **DELETE**d in the same way as they can on the real device. Note that the DEV definitions are global.

Default directories for the DEV device may be set with the **DEV_USE** command.

The DEV device may be redirected with the **DEV_USEN** command.

| Command | Function |
|---|---|
| **DEV_USE** | attach DEV device to a real directory |
| **DEV_USEN** | rename DEV device |

# devices
A device is a piece of equipment on QPC (or the underlying PC) from which data can be received (input) and to which data can be sent (output).

Since the system makes no assumptions about the ultimate I/O (input/output) device which will be used, the I/O device can be easily changed and the data diverted between devices. For example, a program may have to output to a printer at some point during its run. If the printer is not available then the output can be diverted to a Floppy disk file and stored. The file can then be printed at a later date. I/O on QPC can be thought of as being written to and read from a logical file which is in a standard device-independent form.

All device specific operations are performed by individual device drivers specially written for each device on QPC. The system can automatically find and include drivers for peripheral devices which are fitted.

When a device is activated a channel is opened and linked to the device. To correctly open a channel device basic information must sometimes be supplied. This extra information is appended to the device name.

The file name should conform to the rules for a SBASIC type name though it is also possible to build up the file name (device name) as a SBASIC string expression.

In summary the general form of a file name is:

*identifier* [*information*]

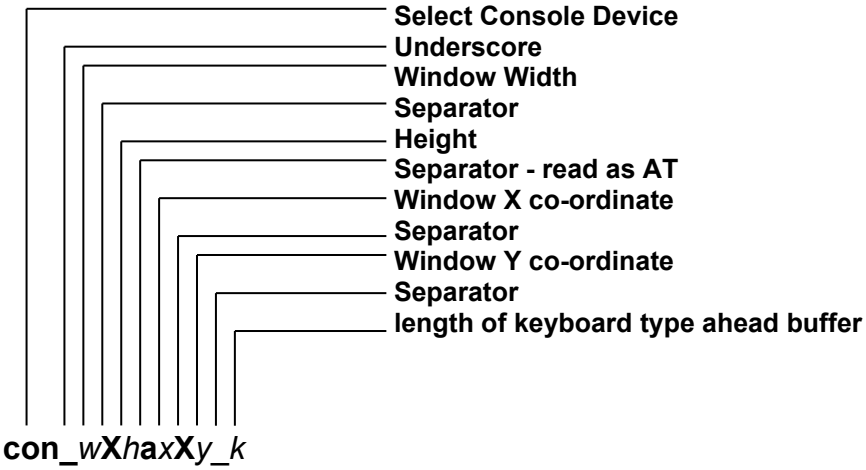
where the complete file name (including the extra information) conforms to the rules for a SBASIC identifier.

Each logical device on the system requires its own particular 'extra information' although default parameters will be assumed in each case where possible.

**define** *device* := *name*

where the form of the device name is outlined below.

**example** for console device




**CON**_*w***X***h***a***x***X***y*_*k*  Console I/O

| *w***X***h* | - window, width, height
| **A***x***X***y* | - window X,Y co-ordinate of upper left-hand corner
| *k* | - keyboard type ahead buffer length (bytes)

default: **con_448x180a32x16_128**

example:
```
OPEN #4,con_20x50a0x0_32
OPEN #8,con_20x50
OPEN #7,con_20x50a10x10
```

**SCR**_*w***X***h***a***x***X***y*  Screen Output

[ *w***X***h* ] - window, width, height
[ **A***x***X***y* ] - window X, Y co-ordinate

default: **scr_448x180a32x16**

example:
```
OPEN #4, scr_0x10a20x50
OPEN #5, scr_10x10
```

**SER***npftce*  Serial (RS-232-C) Receive and Transmit

*n* port number (1, 2, 3 or 4)

[*p*] parity
**e** – 7 bit + even
**o** – 7 bit + odd
**m** – 7 bit + mark (1)
**s** – 7 bit + space (0)

[*f*] handshaking
**i** - ignore flow control
**h** – handshake CTS/DTR
**x** - XON/XOFF

[*t*] translate
**d** - direct output
**t** - translate

[*c*] carriage return
**r** - raw data
**c** - <CR> is end of line
**a** - <CR><LF> is end of line
<CR><FF> is end of page

[*e*] end of file
**f** - <FF> at end of file
**z** – CTRL Z at end of file

default: **ser1htr** (8 bit no parity with handshake, translate)

example:
```
OPEN #3, ser1e
OPEN #4, serxdc
COPY flp1_test_file TO ser1c
```

**SRX***npftce* Serial (RS-232-C) Receive only

*n* port number (1, 2, 3 or 4)

[*p*] parity
**e** – 7 bit + even
**o** – 7 bit + odd
**m** – 7 bit + mark (1)
**s** – 7 bit + space (0)

[*f*] handshaking
**i** - ignore flow control
**h** – handshake CTS/DTR
**x** - XON/XOFF

[*t*] translate
**d** - direct output
**t** - translate

[*c*] carriage return
**r** - raw data
**c** - <CR> is end of line
**a** - <CR><LF> is end of line
<CR><FF> is end of page

[*e*] end of file
**f** - <FF> at end of file
**z** – CTRL Z at end of file

default: **srx1htr** (8 bit no parity with handshake, translate)

example:
```
OPEN_IN #3, srx1e
OPEN #4, srxxdc
COPY srx1c TO flp1_test_file
```

**STX***npftce* Serial (RS-232-C) Transmit only

*n* port number (1, 2, 3 or 4)

[*p*] parity
**e** – 7 bit + even
**o** – 7 bit + odd
**m** – 7 bit + mark (1)
**s** – 7 bit + space (0)

[*f*] handshaking
**i** - ignore flow control
**h** – handshake CTS/DTR
**x** - XON/XOFF

[*t*] translate
**d** - direct output
**t** - translate

[*c*] carriage return
**r** - raw data
**c** - <CR> is end of line
**a** - <CR><LF> is end of line
<CR><FF> is end of page

[*e*] end of file
**f** - <FF> at end of file
**z** – CTRL Z at end of file

default: **stx1htr** (8 bit no parity with handshake, translate)

example:
```
OPEN_NEW #3, stx1e
OPEN #4, stxxdc
COPY flp1_test_file TO stx1c
```

**PAR***ntce* Parallel Port (transmit only)

*n* port number (1, 2, 3 or 4)

[*t*] translate
**d** - direct output
**t** – translate

[*c*] carriage return
**r** - raw data
**c** - <CR> is end of line
**a** - <CR><LF> is end of line
<CR><FF> is end of page

[*e*] end of file
**f** - <FF> at end of file
**z** – CTRL Z at end of file

default: **par1tr** (translate, raw data)

example:
```
OPEN_NEW #3, par1da
OPEN #4, ser
COPY flp1_test_file TO par1
```

**PRT** Printer port (either SER or PAR)

default: none

example:
```
OPEN_NEW #3, prt
COPY flp1_test_file TO prt
```

**NUL***t* Nul device, throw away output, provide dummy input

[*t*] type

**p** – waits (forever or until the specified timeout) on any input or output operation

**f** - emulate a null file. Any attempt to read data return an End of File Error as will any file positioning operation. Reading the file header will return 14 bytes of zero (no length, no type ).

**z** - emulate a file filled with zeros. The file position can be set to anywhere. Reading the file header will return 14 bytes of zero (no length, no type).

**l** - emulate a file filled with null lines. The file appears to be full of the newline character (10). The file position may be set to anywhere. Reading the file header will return 14 bytes of zero (no length, no type ).

default: **nul**

example:
```
OPEN #7, nulz
OPEN #4, nuli
COPY ser1 TO nul
```

**PIPE**_*name*_*l* Two ended Pipe device (first in, first out)

*name* pipe name

[*l*] indicates pipe length in bytes (default 1024 bytes)

default: no default

example:
```
OPEN_NEW #7, pipe_alpha
OPEN_NEW #4, pipe_beta_2048
OPEN_IN #5, pipe_beta
```

**HISTORY**_*name*_*l* Single ended Pipe device (last in, first out)

[*name*] public history name

[*l*] indicates pipe length in bytes (default 1024 bytes)

default: no default

example:
```
OPEN #7, history
OPEN #4, history_messages_2048
OPEN #5, history_512
```

**DEV***n*_*name* Defaulting file accessing device

*n* - Dev drive number

*name* - Dev drive file name

default: no default

example:
```
OPEN #9, dev1_data_file
OPEN #9, dev5_test_program
COPY dev2_test_file TO scr_
```

**FLP***n*_*name* Floppy drive File Access

*n* - Floppy drive number

*name* - Floppy drive file name

default: no default

example:
```
OPEN #9, flp1_data_file
OPEN #9, flp1_test_program
COPY flp1_test_file TO scr_
```

**RAM***n_name* RAM (virtual) drive File Access

*n* - RAM drive number
*name* - RAM drive file name

default: no default

example:
```
OPEN #9, ram1_data_file
OPEN #9, ram1_test_program
COPY ram1_test_file TO scr_
```

**WIN***n_name* Winchester hard disk drive File Access

*n* - WIN drive number
*name* - WIN drive file name

default: no default

example:
```
OPEN #9, win1_data_file
OPEN #9, win1_test_program
COPY win1_test_file TO scr_
```

| Keyword | Function |
|---|---|
| **OPEN** | initialise a device and activate it for use |
| **CLOSE** | deactivate a device |
| **COPY** | copy data between devices |
| **COPY_N** | copy data between devices, but do not copy a file's header information |
| **EOF** | test for end of file |
| **WIDTH** | set width |

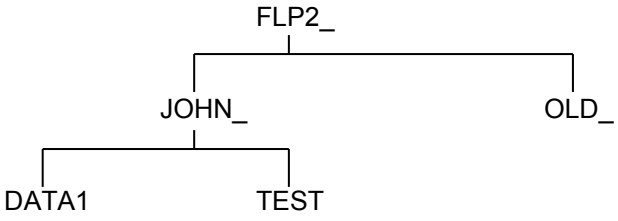
## direct command

SBASIC makes a distinction between a statement typed in preceded by a line number and a statement typed in without a line number. Without a line number the statement is a direct command and is processed immediately by the SBASIC command interpreter. For example, **RUN** is typed in on the command line and is processed, the effect being that the program starts to run. If a statement is typed in with a line number then the syntax of the line is checked and any detectable syntax errors reported. A correct line is entered into the SBASIC program and stored. These statements constitute a SBASIC program and will only be executed when the program is started with the **RUN** or **GOTO** command.

Not all SBASIC statements make sense when entered as a direct command, for example, **END FOR**, **END DEFine**, etc

## directories

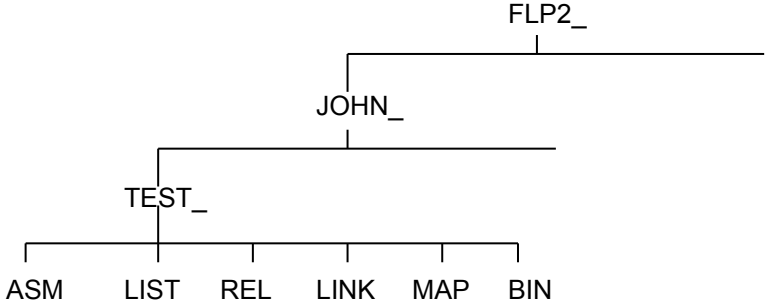
In SMSQ terminology, a 'directory' is where the system expects to find a file. This can be as simple as the name of a device (e.g. FLP2_ the name of floppy disk drive number 2) or be much more complex forming part of a 'directory tree'.
For example: the directory FLP2_ could include directories JOHN_ and OLD_ (note: all directory names end with an '_'), and JOHN_ could include files DATA1 and TEST).




This shows another characteristic of the 'directory tree': it grows downwards. The complete SMSQ filename for DATA1 in this example is FLP2_JOHN_DATA1. (You may have come across the terms 'pathname' or 'treename': these refer to the same thing as a SMSQ filename.)

One unusual characteristic of the SMSQ directory structure is the absence of a formal file name 'extension'. This is not strictly necessary as 'extensions' (e.g. _aba for ABACUS files, _asm for assembler source files etc.) are treated as files within a directory.

This can be illustrated with the case of an assembler program TEST, processed using the GST macro assembler and linkage editor. The assembler source file (TEST_ASM), the listing output from the assembler (TEST_LIST), the relocatable output from the assembler (TEST_REL), the linker control file (TEST_LINK), the linker listing output (TEST_MAP) and the executable program produced by the linker (TEST_BIN) are all treated as files within the directory TEST_.



SMSQ/E provides facilities to set default directories. The defaults are available for all filing system operations. A default may be set to any level of complexity and gives a starting point for finding a file in the tree structure. Thus, in this example, if the default is FLP2_, then JOHN_TEST_ASM will find the assembler source. If the default is FLP2_JOHN_, then TEST_ASM will find it, while the full filename FLP2_JOHN_TEST_ASM will find the file regardless of the default.

| Command | Function |
|---|---|
| **DATA_USE** | set data default used by **LOAD**, **OPEN** etc |
| **PROG_USE** | set program default used by **EX**/**EXEC** etc |
| **DEST_USE** | set destination default used by **COPY**, **RENAME** etc |

## directory devices

Directory devices handle individual files, organised in directories (with at least one root directory). The drive RAM is used to access the RAM-disk, FLP is used to access the floppy disk, and WIN is used to access the hard disk. More details can be found in the hardware-dependent sections of this manual. SMSQ/E will read and write from and to QL floppy disk (DD and HD, if your hardware permits).

In addition, SMSQ/E comes with in built drivers to recognise, (PC) DOS floppy disks, and (Atari) TOS floppy disks (DD and HD).

The SBASIC command **DIR** has been extended to show density and format of a medium. There are now new functions, which allow you to fetch this information, see the **DMEDIUM_xxx** range of functions.

If you insert a QDOS 720k floppy disk into flp1_ and type:

```
DIR flp1_
```

Then you will see the following (or similar) output on the screen:

```
diskname QDOS DD
720/1440 sectors
...directory ...
```

If you insert a DOS high-density disk and ask for the directory again, you should see:

```
DISKNAME MSDOS HD
720/2880 sectors
...directory ...
```

# DOS
# device

The DOS device has been created to transfer data between the Windows and the SMSQ/E environment. Using the device you can directly browse your PC hard disks (or network drives or CD-ROMs or whatever), read and write files.

Please note that the DOS device is NO replacement for the WIN device (it never was intended to be), all SMSQ header information gets lost on DOS drives, therefore you cannot store executable code on them.

You can use this device in the same way as any other QPC directory device to access and exchange files between Windows and SMSQ/E as easy as never before. The usual restrictions imposed by the general QDOS file naming convention apply, i.e. the length of the directory + filename is limited to 36 characters. Names longer than that won't show up in the directory lists! Therefore, it is a good idea to place files, which you want to access from both SMSQ/E, and Windows only one or two directory levels deep or change the base of one DOS drive directly below the desired directories.

Many filenames that are valid under SMSQ/E are not valid on Windows. The offending characters (e.g. *, /, ? etc. or filenames with spaces at their end) are translated into other, valid ANSI characters. This conversion works quite well, but you are advised to use valid filenames wherever possible.

One problem with the SMSQ/E way of accessing files is that the "_" separator can be a valid part of a name or a directory separator. Therefore the relation SMSQ/E filename -> Windows filename is ambiguous. This can cause some problems:

Let's say you have two directories named **C:\QL\STUFF** and **C:\QL\STUFF_NEW** and you want to create a file called **DOS1_QL_STUFF_NEW_BRANDNEW.TXT**. Where does that file belong? It could mean any of the following choices:

**C:\QL_STUFF_NEW_BRANDNEW.TXT**
**C:\QL\STUFF_NEW_BRANDNEW.TXT**
**C:\QL\STUFF\NEW_BRANDNEW.TXT**
**C:\QL\STUFF_NEW\BRANDNEW.TXT**

Probably the last one is the one you intended it to be, but how should QPC now? The easy solution is not to use underscores in directory names. But if you can't help it, it gets essential to know how the DOS device works.

Since v3.02 there is a new algorithm which is based on the simple assumption that if you have a directory called "**QL_STUFF**" you won't also create "**QL\STUFF**".

The basic principle is that the algorithm always searches for the longest consecutive parts of the name. In the above example QPC would begin with searching for any directory starting with "**C:\QL**". If there is none the process is complete and the result is simply "**C:\QL_STUFF_NEW_BRANDNEW.TXT**". Otherwise it will look for any directory starting with "**C:\QL_STUFF**" next. Again, if there is one, QPC will try "**C:\QL_STUFF_NEW**" and so on. If not found, however, it will test whether the last successful part ("**C:\QL_STUFF**") is itself a directory. If it is, it is considered as a part of the filename and all future searches use it as their base (i.e. next step being "**C:\QL_STUFF\NEW**"). If not the search terminates with the result again being "**C:\QL_STUFF_NEW_BRANDNEW.TXT**".

If this sound too confusing or too badly explained (probably both) just remember one thing: never use "_" within directory names.

Finally please note that you cannot use **RENAME** to rename files on a DOS drive. SMSQ/E allows you to rename files from one directory to another one, which is not compatible with the DOS way of renaming files. If you want to rename a file, you need to **COPY** it to a new location and **DELETE** the old file.

## DOS disks

You can load files from (PC) DOS disks as if they were QPC disks. You can save files to DOS disks, but you have to make sure that the filename does match the DOS naming convention, i.e. up to eight characters, full stop, up to three characters for the extension.

All the filling system calls will work on DOS disks, you can create subdirectories, delete files. You cannot, however, use the **FORMAT** command to format a floppy disk to DOS format. It will always be the preferred (QDOS) format.

The DOS filling system does not have the concept of different file types. Different file types are distinguished by their filename extension. Therefore, QDOS "executable" programs (file type 1) cannot be handled the way they are handled on a QDOS disk. From SMSQ/E version 2.87 on, you can copy executable files onto DOS disks, which can later be executed from this disk. They will get a special extension '.EXn' where n is the number which specifies the dataspace (which is usually held invisible to the user in the file header): it is 512*2^n. This extension will be invisible in SMSQ/E, but will be seen in DOS. Example (assuming flp1_ contains a DOS disk):

**COPY win1_CLOCK TO flp1_CLOCK**

Will create a file ***flp1_CLOCK.EX1*** on the DOS disk. You can still refer to it as flp1_CLOCK, it will be shown in the directory as flp1_CLOCK only, but if you look at this disk on a DOS computer, then you will see the real name. Extensions of executable files will be removed automatically, e.g.

**COPY win1_PROGRAM_bin TO flp1_PROGRAM.bin**

Will not create a file ***flp1_PROGRAM.bin***, it will create a file ***flp1_PROGRAM.EX3***, but you have to refer to it as flp1_PROGRAM only, e.g.

**EX flp1_PROGRAM**

As the filename extension is lost anyway even if you copy the file back, we suggest that you do not specify an extension. This will also make sure that you do not end up with files having the same filename.

# error handling

Errors are reported by SBASIC in a standard form:

**At line** *line_number* **:** *statement_number error_text*

Where the line number is the number of the line where the error was detected, statement number is the number of the statement in the line, and the error text is listed below.

**(1) incomplete**
An operation has been prematurely terminated (or break has been pressed).

**(2) invalid job ID**
An error return from SMSQ/E relating to system calls controlling multitasking or I/O.

**(3) insufficient memory**
SMSQ/E and/or SBASIC has insufficient free memory.

**(4) value out of range**
Usually results from attempts to write outside a window or an incorrect array index.

**(5) buffer full**
An I/O operation to fetch a buffer full of characters filled the buffer before a record terminator was found.

**(6) invalid channel ID**
Attempt to read, write or close a channel which has not been opened.
Can also occur if an attempt to open a channel fails.

**(7) not found**
File system, device, medium or file cannot be found.
SBASIC cannot find an identifier. This can result from incorrectly nested structures.

**(8) already exists**
The file system has found an already existing file with the same name as a new file to be opened for writing.

**(9) is in use**
The file system has found that a file or device is already exclusively used.

**(10) end of file**
End of file detected during input.

**(11) medium is full**
A device has been filled (usually Floppy disk).

**(12) invalid name**
The file system has recognised the name but there is a syntax or parameter value error.
In SBASIC it means a name has been used out of context. For example, a variable has been used as a procedure.

**(13) transmission error**
RS-232-C parity error

**(14) format failed**
Attempted format operation has failed, the medium is possibly faulty (usually a Floppy disk).

**(15) invalid parameter**
There is an error in the parameter list of a system or SBASIC procedure or function call.
An attempt was made to read data from a write only device.

**(16) check failed**
The medium (usually a Floppy disk) is possibly faulty

**(17) error in expression**
An error was detected while evaluating an expression.

**(18) arithmetic overflow**
Arithmetic overflow division by zero, square root of a negative number, etc.

**(19) not Implemented**

**(20) write protected**
There has been an attempt to write data to a shared file.

**(21) invalid syntax**
A SBASIC syntax error has occurred.

**(22) PROC/FN cleared**
This is a message which is for information only and is not reporting an error. It is reporting that the program has been stopped and subsequently changed forcing SBASIC to reset its internal state to the outer program level and so losing any procedure environment which may have been in effect.

**(23) access denied**

**error reporting**
The line number where an error occurred, is returned by **ERLIN.** And the error number by **ERNUM**.

**REPORT** will report the description of the last error encountered.

**ERT** can be used with functions which return an error code, in order to allow the program to stop, or continue.

**error recovery**
After an error has occurred the program can be restarted at the next statement by typing

**CONTINUE**

If the error condition can be corrected, without changing the program, the program can be restarted at the statement, which triggered the error. Type

**RETRY**

**error handling**
Error handling is invoked by a **WHEN ERROR** clause. When an error is encountered, processing is passed to the commands in the **WHEN ERROR** clause. Within the **WHEN ERROR** clause the type of error can be tested for, and appropriate actions can be taken.

# expressions

SBASIC expressions can be string, numeric, logical or a mixture: unsuitable data types are automatically converted to a suitable form by the system wherever this is possible.

**define**
*monop* := | **+**
| **-**
| **NOT**

*expression* := | [*monop*] *expression operator expression*
| (*expression*)
| *atom*

*atom* := | *variable*
| *constant*
| *function* [ (*expression* *|, *expression* *)]
| *array_element*

*variable* := | *identifier*
| *identifier*%
| *identifier*$

*function* := | *identifier*
| *identifier*%
| *identifier*$

*constant* := | *digit* * [*digit*] *
| *[*digit*] *, *[*digit*]*
| *[*digit*] * [.] *[*digit*]* **E** *[*digit*]*

The final value returned by the evaluation of the expression can be integer giving an **integer_expression**, string giving a **string_expression** or floating point giving a **floating expression**. Often floating point and integer expressions are equivalent and the term **numeric_expression** is then used.

Logical operators can be included in an expression. If the specified operation is true then a one is returned as the value of the operation. If the operation is false then a zero is returned. Though logical operators can be used in any expression they are usually used in the expression part of an **IF** statement.

example: i. **test_data + 23.3 + 5**
ii. **"abcdefghijklmnopqrstuvwxyz"(2 TO 4)**
iii. **32.1 * (colour = 1)**
iv. **count = -limit**

# Extended Environment

## The Parts of the Extended Environment

The Extended Environment comes as four loosely connected parts. The Pointer Interface, the Thing System, the Window Manager, and the HOTKEY System 2.

The Pointer Interface is an extended version of the QDOS CONsole driver which accepts user input from a "pointing device", usually a "mouse" as well as from the keyboard. The user's input is directed to the program that he wishes to use by pointing to that program with a "pointer" (an arrow or other pointing symbol which appears on the screen). The Pointer Interface also keeps the display tidy when there is more than one Job trying to write to the display.

Programs do not have to be written specially for the Pointer Interface; all the window save and restore operations are done automatically.

The Window Manager is a set of utility routines that provides menu handling facilities to programs which have been written specially for the Extended Environment. These facilities create a user interface, which is reasonably uniform and consistent from program to program, even where these programs come from completely different suppliers.

The HOTKEY System 2, in contrast, is entirely under the user's control. A Hotkey can be used to Execute a program, to pick a one of the Jobs executing so that you can work with it, to stuff pre-defined strings into the keyboard queue, to recall the last line typed or to transfer strings from one program to another.

The standard system incorporates SBASIC functions to add and remove Hotkeys, but, as all the operations required to control the HOTKEY System 2 are built into the Hotkey Thing (*Thing!! of all the parts of the Extended Environment, Things are the simplest and most confusing to the uninitiated*), there is no problem in providing the same control through other programs. QPAC 2, for example, provides some facilities for users to access the HOTKEY System 2.

The Thing System is something, which most users do not need to bother themselves with. The Thing System exists to make it much easier for software developers to write programs which communicate cleanly with programs from other suppliers. There is no direct user control over the Thing System, but for those who might be interested, here is some background information.

*SMSQ/E allows Jobs to communicate directly with each other without the need to pass the information through "pipes". They can do this by sharing some area of the computer's memory. To maintain the self-cleaning aspects of the SMSQ operating system, the shared memory and the communicating Jobs will normally need to be owned by the same Job. If this owner Job, is removed, the communicating Jobs will be removed as well as the shared memory. Thus the system is kept clean.*

*The Thing System is a means of defining and controlling areas of shared memory. A Thing can be used to transfer data between Jobs, which are completely independent, and these Jobs do not need to be executing at the same time. A Thing can be permanent until removed, or it can be owned by a Job. When a Job owning a Thing is removed from the computer, then the Thing is also removed by SMSQ/E at the same time. If there are any Jobs using a Thing, which is removed from the computer, then the Jobs will be removed.*

They are called Things because they can be almost anything. The Window Manager should be a Thing, but it was written before Things were implemented. HOTKEY System 2 is a Thing. The programs, which form QPAC 2 are Executable Things. Things can be used to control access to sound synthesizers, to control the layout of the display (e.g. the Button Frame of QPAC 2), to provide functions which can be accessed from any programming language (e.g. FILE_SELECT$ from Jochen Merz's MENU extensions) or almost anything else.

## The Pointer Interface

The main purpose of the Pointer Interface is to allow the user to organise the display in such a way as to make it easier to see and control a large number of Jobs running in the computer at any time.

The Pointer Interface is an extended QDOS CONsole driver, which accepts user input from a "pointing device", usually a "mouse", as well as from the keyboard. And the user's input is directed to the Job that he wishes to use by pointing to that Job's windows with a "pointer" (an arrow or other pointing symbol, which appears on the screen).

This would not be very useful if there were many Jobs with windows which overlapped in the usual confused way, as it would be impossible to tell which of these Jobs was the intended recipient of the user's instructions. To avoid this problem, the Pointer Interface ensures that, when you have two or more Jobs with overlapping windows, the windows belonging to one of these Jobs will appear to be on top of the other Jobs' windows. It is the top Job that will get the user's keystrokes and mouse button presses.

If the top Job will not accept Pointer input, but is waiting for the user to type on the keyboard. Then the pointing symbol will change to a letter K. If, for any reason, the top Job will not accept any entry from the keyboard or mouse, the pointing symbol will change to a "No Entry" sign.

## The Cursor Keys and the Mouse

The cursor keys can usually be used to move the pointer around the display in place of using the mouse. You may find that it is easier to work with some programs if this facility is suppressed. There are two commands, which can be used to turn the cursor key control off and on again.

**CKEYOFF** *Turn cursor key control off*
**CKEYON** *and back on again*.

The mouse can also be used to generate cursor key strokes, for programs which accept cursor key input, by pressing the left hand button and moving the mouse while holding it down.

## Locked Windows

To keep the screen tidy, if a Job's windows are partly or completely covered by another Job's windows, then the lower Job's windows are "locked". If a Job tries to write to a locked window, or to read keystrokes through a locked window, then it will be suspended by SMSQ/E (not by the Pointer Interface). If the pointer is over a locked window, the pointing symbol changes to a picture of a padlock. The padlock will also appear if the display is frozen (e.g. by pressing CTRL F5).

## Picking

There are various ways in which a Job can be "Picked", either to unlock its windows so that the Job can write to them, or to direct the keyboard queue to that Job.

If part of a Job is visible, then you can point to the Job and Pick it by pressing the left hand mouse button or space bar. You can also use the right hand button or ENTER key, but this also generates a "Wake Event" (see under "Events").

It is possible to bring the Job at the bottom of the pile of windows to the top by pressing CTRL and C (the standard SMSQ/E keyboard switching keystroke).

Finally, a specific Job can be Picked to the top by any Job written to use the Pointer Interface extended operating system entry point IOP.PICK. In particular, Jobs can be Picked using the Pick program in QPAC 2 or using a Hotkey through the HOTKEY SYSTEM 2.

## Unlockable Windows

It is possible to define a Job's windows to be "unlockable". These windows are kept outside the control of the Pointer Interface and thus are every bit as badly behaved as the standard QDOS CONsole driver windows. This means that, for example, it is possible to create a clock program which writes the time and date into a spare hole in the Quill display. Unfortunately, it is unlikely that other programs will have a spare hole in exactly the same place and such a clock program is of limited use as it will make rather a mess of any other programs which are used at the same time as Quill.

The HOTKEY System 2 includes facilities for executing programs with unlockable windows.

*There is a clean solution to this problem. If you wish to have two Jobs writing to the same set of windows, so that both Jobs are locked and unlocked together, the window used by the clock (or other similar program) should be owned by the other Job. This is the solution used by the Calendar program in QPAC 1. If you are writing a program in SBASIC, and you wish to have a clock linked to the program, then you can do this by opening a window where you wish the clock to be, and executing the clock in that window.*

## Primary and Guardian Windows

It is common for Jobs to have more than one window open at a time. To keep things simple, the Pointer Interface defines a "Primary Window" area. The Primary Window area of each Job is used to determine which Jobs overlap on the display.

For true Pointer Environment Jobs, the Primary Window is defined by a special operating system call, and, although the Primary Window can be moved or re-sized, other windows owned by the Job must be within the Primary Window area. For other programs it is just the smallest rectangular area which encloses all of the Job's windows. If these windows change in size or position, then the primary window may change as well. In turn, this may cause parts of other Job's windows to be restored or even unlocked. This may be desirable (e.g. where a modest Job is being moved around the display) or it can have some unpleasant effects.

To prevent a Job's Primary Window changing size, it is possible to define a "Guardian Window" which is opened for the Job, before the Job itself starts executing. The HOTKEY System 2 can be used to execute programs with Guardian Windows. A Guardian Window will be the Primary Window for the Job, and must be defined to cover the whole of the area of the display that will be used by the Job. In most cases this Guardian Window will cover the whole of the Display.

## Restoring Windows

When a Primary Window is locked because another Primary Window is opened, expanded or moved on top of it, the whole Primary Window area is saved.

When part or all of a Job's Primary Window becomes visible as a result of other Primary Windows being moved, Primary Windows being closed or the Job itself being Picked, the Pointer Interface will restore those parts of the Primary window that have become visible. Unlike most window environments, where it is the responsibility of each Job to maintain its own windows, this is done by the Pointer Interface without any co-operation from the Job. This method, therefore, works as well with Jobs that are written in ignorance of the Pointer Interface as it does with Jobs that are written to take advantage of it

## Events

To improve the efficiency of the system, the Pointer Interface provides a "Event Vector" for programs using the Pointer. These programs do not need to keep on checking the position of the Pointer or the state of the mouse buttons. Instead, these programs will suspend themselves until an Event occurs, such as the Pointer moving into the window or a key or mouse button being pressed. It is up to each program to interpret these Events. The Window Manager provides a uniform response to these Events.

## The Pointer Interface and Badly Behaved Software

In principle, any well written software can be "multitasked" under SMSQ/E without any additional software and without any real problems. All you need to do is to **EXEC** the programs you wish to use (**EXEC**ing new programs and quitting the ones already executing whenever you wish). When you wish to direct the keyboard input to another Job, just press CTRL and C until you get to it. The screen can get a little untidy, but using the Pointer Interface can cure that.

In practice, there are several problems, one of which can occur in both well written and badly written software.

The SMSQ/E CONsole driver assumes that all Jobs requiring input from the user will have an active keyboard cursor. This is not necessarily so in keyboard based graphics programs which bypass the CONsole driver and read the keyboard directly. (This enables them to detect cursor key presses at the same time as other keystrokes, a facility not available through the CONsole driver.) As these direct keyboard reads bypass the CONsole driver, it is not possible to stop such Jobs reacting to keystrokes intended for other Jobs. It is possible to give this type of job a Guardian Window with a "Freeze" option. When the Job is buried by another Job, the program is suspended, thus preventing it from stealing the input intended for another program.

There is a similar, but less serious, problem with menu based programs which read the keystrokes through a CONsole which does not have a cursor. Although the Psion programs have a visible cursor, they use the trick of reading though a special CONsole which does not have a cursor. Thus, without the Pointer Interface you cannot use CTRL C to switch the keyboard input to one of these menu based programs or one of the Psion programs. With the Pointer Interface you can use CTRL C to switch the keyboard input to any Job which reads the keyboard through the CONsole driver.

*So much for the type of problems for which there is sometimes some justification, now we come to the nasty bits.*

In addition to their suppression of CTRL C, the Psion programs have a number of other nasty tricks to play on unsuspecting users. The first has nothing to do with the display: when Quill, or one of the other Psion programs starts executing, it grabs most of the available memory. It may not need all this memory, but doing this ensures that there is not enough memory left for any other program to be executed. It also means that there is not enough memory for a window save area to be allocated by the Pointer Interface. The HOTKEY System 2 includes facilities to execute the Psion programs in such a way that the amount of memory they use is limited.

The next nasty trick is that the Psion programs make a large number of unnecessary operating system calls to set the display MODE. The Pointer Interface can survive this, but it can be rather annoying for the user.

Their worst trick is writing directly to the display. In the case of the Psion programs, this trick performs no useful function and the harmful side effects can be minimised by the use of a Guardian window. There are programs which are much worse, but even so, some of these will work if they are executed within a Guardian window with the Freeze option which will stop them writing to the display (or doing anything else!) while they are buried.

There are two types of program which are sure to give problems. The first is the type of program which is so badly behaved that it is unable to share the machine with another copy of itself. The second is the type of program which pokes values directly into the operating system data structures.

## The Window Manager

The Window Manager provides a set of utility routines, which simplify the handling of menus and pull-down windows. There is no reason why any particular program should use the window manager, but using it provides a reasonably uniform user interface to applications programs. The following description applies to standard menu windows. Other types of windows should be similar, but there may be some subtle differences.

A window set up by the window manager has a number of different parts. The first is general information: this could be lines or borders dividing the window to improve the clarity, or explanatory text or icons. Then there are the "loose menu items": so called because they are not tied down to any fixed organisation and can be put anywhere in the window. Finally, there are the "sub-windows" (*These are called "application sub-windows" within the Window Manager to distinguish them from the "information sub-windows" which just contain the general information*). Simple pull-down menus may not have any sub-windows, whereas in menus that include lists of items to select (e.g. lists of files) the sub-window may well be the most important part.

## Selecting Menu Items

A menu comprises a number of "items". Items can be selected by pointing to the item using the cursor keys or mouse, and then pressing the space bar, ENTER or one of the mouse buttons. The space bar is equivalent to the left button: pressing this is termed a "HIT". The ENTER key is equivalent to the right button: pressing this is called a "DO".

Items have one of three states: "unavailable", "available" or "selected". Most items will be available, selected items are shown using a highlight colour combination, while unavailable items are shown in a reduced visibility colour combination.

When you point to an item, a border will appear around it: this indicates that it is the "current item". This is the item which will be HIT or DONE. If the item is unavailable, then there will be no effect. If the item is available, then the item will be set to the selected state, and the action associated with the item will be carried out. If the item is selected, then a HIT will make the item available again, while a DO will keep the item selected and then carry out the associated action.

The distinction between a HIT and a DO depends on the operation. Very often a HIT will merely change the state of the item, while a DO will do the action. For other items HIT and DO are similar but DO is more forceful. For some items there is no distinction between them.

You can select several items in a sub-window menu by holding the mouse button down and moving (slowly) through the menu. As the pointer moves into each item, it will be selected.

## Single Keystroke Selection

It is also possible to select items using a single keystroke. This key will often be the first letter of the item name (if it has a name), it may be shown in the menu as a symbol close to the item, or you may need to memorise it from the manual.

The effect of single keystroke selection also depends on the operation. For an item within a sub-window, it may just move the pointer to the item, making it the current item, or it may move it to the item and HIT or DO it. For Loose menu items, it will HIT or DO the item without moving the pointer: this stops the pointer being moved out of the current sub-window.

Current sub-window? Yes, the Window Manager allows there to be several sub-windows in one window. Usually, items in a sub-window can only be selected by single keystroke if the pointer is in that sub-window. So that keystroke selection of sub-window menu items can be used from anywhere in the window, each sub-window has a selection keystroke. To move the pointer into the sub-window, press the sub-window selection keystroke: you can then select items within the sub-window by single keystroke.

## Pan and Scroll

Items in a menu sub-window are arranged in rows and columns. There may be more items in a menu sub-window than can be displayed in the available space. If this happens, the window will be marked as pannable (you can move the contents sideways), or scrollable (you can move the contents up and down). The window can be both pannable and scrollable, but this is usually very inconvenient.

A window is marked as pannable or scrollable by including rows of arrows within the window, or by putting a pan or scroll bar to the right of or below the window. The pan or scroll bar includes a block which indicates the (size and) position of the visible section of the menu within the complete menu. If the block in a scroll bar starts halfway down the scroll bar, then the first visible row in the menu is about half way down the list or rows in the complete menu.

To pan or scroll by one column or row, you can press ALT and a cursor key. HITting an arrow row will have the same effect. To pan or scroll by the width (less one column) or height (less one row) of the window, you can press ALT, SHIFT and a cursor key. DOing an arrow row will have the same effect.

To pan or scroll directly to a position in the menu, move the pointer to the pan or scroll bar and HIT the bar at the position required. If you keep your finger on the left button or the space bar, you can "drag" through the menu using the mouse or cursor keys. To make this simpler for keyboard users, the sub-window selection keystroke will also move the pointer from inside the sub-window to the pan or scroll bar.

## Split and Join

Some menu sub-windows can be split into two or more sections. Each section can then be panned or scrolled independently. To split a window, point to the pan or scroll bar where you wish to split to be made and DO it. The window may be re-joined by DOing on the split.

## Standard Loose Items

There are a number of standard loose menu items and keystrokes. The most common are the F1 (Help), F3 (Command) and ESC (Escape - leave the menu). As these are the most common, they also tend to be the most variable. Help may be available using F1, even if there is no Help item visible. F3 or the command item will usually give access to a further (pull-down) menu. ESC can mean leave the menu, leave the menu without saving any changes made or leave the program altogether. If in doubt, try it.

There is a set of window control items which are all selected by **CTRL** and a function key (**F1** to **F4**). These keystrokes are denoted **CF1**, **CF2**, **CF3** and **CF4**. The operations will only be available if the appropriate item is available.

**CF4** Move
**CF3** Resize
**CF2** Sleep
**CF1** Wake

To move the window, HIT or DO the Move item or press **CTRL** and **F4**. The window can then be moved using the mouse or cursor keys. The pointing symbol changes to the Move symbol, and it may happen that the only thing that moves while you are positioning the window is this symbol. When you have moved far enough, HIT or DO will complete the operation.

Resize is similar to move: HIT or DO the Resize item, or press **CTRL** and **F3**, and you can then change the window size.

The purpose of the Sleep item is to tuck the Job away to bed to free some of the display. This is especially useful for Jobs with large windows. The Job will shrink its window to a small "button", and wait until it is woken up.

The Wake item is used to create a "wake event" to the Job. If the item is a button, the wake event will make it re-create its windows. If the windows are already set up, the wake event will make the Job refresh the menus. This is useful for Jobs, which display information about the system. The Files menu of QPAC 2, for example, will re-read the directory of the appropriate disk when it receives a wake event.

A wake event is also created when a Job is picked by a DO rather than a HIT, and it is possible to send a Job a wake event using the HOTKEY System 2.

## The HOTKEY System 2

The HOTKEY System 2 provides Hotkey facilities. A Hotkey is a key, which is pressed to cause an action, which is independent of the program with which you are working at the time. For example, if you have a Hotkey, which pops up a telephone directory, then it does not matter whether you are in the middle of using a word processor, or doing your accounts, you press the Hotkey and up pops the directory. The keystroke is stolen by the HOTKEY System 2, and so the program you are working with, remains blissfully unaware that anything has happened.

Using HOTKEY System 2, the ALT key is used to indicate that a keystroke may be a Hotkey. This operates in the same way as the CTRL and SHIFT keys and may be used in combination with either (or both) to define up to 128 Hotkeys.

## Hotkey Operations

There are many different operations which may be carried out using Hotkeys. To make it even more flexible, there is one operation that allows any code to be added to HOTKEY System 2. In this manual, however, we will describe only those operations which can be set up using the SBASIC extensions which are incorporated into HOTKEY System 2.

Three Hotkeys are set up with the HOTKEY System 2. These are intended to save you time and effort by doing some typing for you.

| | |
|---|---|
| **ALT ENTER** | recalls the last line you typed into the current keyboard queue (and the line before that, and so on). |
| **ALT SPACE** | copies the current " Stuffer Buffer" into the current keyboard queue. |
| **ALT SHIFT SPACE** | copies the previous " Stuffer Buffer" into the current keyboard queue. |

(The file name is put into the Stuffer Buffer by QPAC 2 when a file is Viewed or by QD when a file is saved. Other programs put whatever they wish into the Stuffer Buffer, and you can set the Stuffer Buffer within your own SBASIC programs with the **HOT_STUFF** command. You can read it programatically using the **HOT_STUFF$** function)

You can also set up Hotkeys to copy predefined strings into the current keyboard queue. This can be useful for common phrases such as "Yours sincerely" or long command sequences such as "**F3 P D ENTER N P**" which prints a spreadsheet from Abacus.

The second group of Hotkeys is concerned with executing and Picking programs. Hotkeys can be set up execute programs either from file, or from Executable Things. You can define Hotkeys which will Pick programs which are already executing, and Hotkeys which will Wake programs or Executable Things.

Using one of this second group, you can use a Hotkey to pop up a program you want to use, on top of the program you are using at the time.

## How Hotkey System 2 works

*All the Hotkey operations are performed by a Job called HOTKEY. There is a small task which examines the keyboard queue after a keystroke has been put into it. When you press an ALT key combination which has been set up as a Hotkey, this task will pass a special Event to the HOTKEY Job which will leap into action and do whatever has been specified. If the attempt fails (possibly because there is not enough memory) the HOTKEY Job will burp and retire into the background again. If there is no room in the current keyboard queue, then Hotkeys, like any other keystroke, will get lost.*

## Setting Hotkeys Using SBASIC

Using Hotkeys is very simple, but unfortunately, you do have to set them up first. HOTKEY System 2 includes a number of SBASIC functions to enable Hotkeys to be set up, changed and removed by SBASIC programs. Using functions, instead of procedures, enables error checking to be carried out simply, and any corrective action taken. The HOTKEY System 2 functions (and procedures) start with "**HOT_**" so you should have no problem identifying them. Most Hotkeys will be set up in a BOOT file, but you can add, remove or change any Hotkeys at any time, either by typing the appropriate commands into the SBASIC command console, or by **RUN**ning an SBASIC program.

## Case Dependent Hotkeys

You can define Hotkeys in two ways. If you define a lower case Hotkey, then the Hotkey action can usually be invoked by pressing **ALT** and the appropriate letter, regardless of whether the **SHIFT** key is pressed or **CAPSLOCK** is set. If, however, you define an upper case Hotkey, then this action, will only be invoked by **ALT**, and the upper case character.

For example, if these Hotkeys are set:

| Hotkey | Action |
|---|---|
| a | Execute Alarm |
| Q | Execute Quill |
| q | Execute QRAM |

"**ALT Q**" (**ALT SHIFT Q**) will execute Quill, while "**ALT q**" will execute QRAM. Both "**ALT A**" and "**ALT a**" will execute the alarm clock.

## Summary of Functions to set up Hotkeys

| Function | Sets Hotkey to |
|---|---|
| **HOT_KEY (***key*, *list of strings***)** | copy strings to keyboard queue |
| **HOT_CMD (***key*, *list of commands***)** | send commands to SBASIC |
| **HOT_RES (***key*, *filename***)** | execute resident program |
| **HOT_RES1 (***key*, *filename***)** | ... one copy only |
| **HOT_CHP (***key*, *filename***)** | execute resident program |
| **HOT_CHP1 (***key*, *filename***)** | ... one copy only |
| **HOT_LOAD (***key*, *filename***)** | load and execute program |
| **HOT_LOAD1 (***key*, *filename***)** | ... one copy only |
| **HOT_THING (***key*, *Thing name***)** | execute Thing |
| **HOT_PICK (***key*, *Job name***)** | Pick Job |
| **HOT_WAKE (***key*, *Job name***)** | Wake Job |

## Errors when Setting Hotkeys

The functions used to set up, change and remove Hotkeys have two distinct error handling methods. If the function is used incorrectly, (e.g. missing parameters), then execution of the program will stop in the usual way. If, however, the parameters are correct, but you are trying to do an operation which is not allowed, or proves to be impossible (e.g. redefining a Hotkey without removing it, or trying to load a file, which does not exist), then the function will return an error code. This error code can be used to ask the user (probably yourself) to do some corrective action (e.g. put a particular disk in the drive) before the Hotkey function is called again.

One error code, which can be returned from any of the functions, is ERR.IU (-9, in use). This can occur if another Job has tied up the Hotkey system for more than 2 seconds. If there is a long pause before an "in use" error return, this is the most likely reason.

## Error Reporting

If you do not wish to do any error processing, it would be more convenient to call these functions as procedures. A BOOT (or other program) would then stop automatically with the usual cryptic error messages. Unfortunately this cannot be done directly with the standard SBASIC interpreter, but the Hotkey system includes a simple procedure which will report the error and stop if its parameter value is negative. This procedure, **ERT**, can be used with any function which returns an error code (e.g. many of the Qtyp SPELL functions) as well as with the Hotkey functions. Thus there are three main ways of calling the Hotkey functions:

| | |
|---|---|
| **hkerr = HOT_RES (' t', Qtyp)** | get error code from HOT_RES |
| **PRINT HOT_RES (' t' , Qtyp)** | print error code from HOT_RES |
| **ERT HOT_RES ('t', Qtyp)** | stop and report if error |

**Hotkey Filenames and Other Names**

Some of the functions to set Hotkeys need to be supplied with a file name. You will not usually need to specify a drive name as HOTKEY System 2 will use the Program Default (set up by **PROG_USE**).

In general, the textual parameters of a Hotkey function can be given as either "strings" or "names". A name must start with a letter, and contain only letters, digits and underscores. A string can have any characters between apostrophes or quotes. If in doubt put the parameter between quotes or apostrophes: particularly if you will be compiling your program.

Furthermore, when defining the Hotkey itself, the key is best placed between apostrophes of quotes to avoid problems with the SBASIC name handling which does not distinguish between upper and lower case.

## Boot Programs for the Extended Environment

When the QPC starts up, or after being reset. The SBASIC interpreter will load and run a SBASIC program called "BOOT". This program should be used to set up QPC to match the way you wish to use it. This BOOT program will usually be stored on WIN1_ as WIN1_BOOT, alough a floppy disk can also be used if you configure QPC to boot from a floppy first.

The majority of QL/QPC software falls into one of two main groups, "resident extensions" and "executable programs". The other important group, SBASIC programs. SBASIC programs compiled with QLiberator or Turbo are true "transient programs".

"Resident extensions" are provided to expand the capabilities of QPC. Some are supplied on disk and need to be loaded in at the start of a session and remain resident in QPC for the whole of that session. The Extended Environment comes built into SMSQ/E and does not need to be loaded. Other typical examples are the Pointer Toolkit and the Spell extensions. All of these are intended to be of use for many different programs throughout an entire session.

"Executable programs" are designed to come and go as required. These are executed as required, and when you have finished with them, they go away, leaving QPC's memory free for other executable programs. Typical examples are Quill, Abacus and the other Psion programs.

Some executable programs require specific resident extensions to be present. The reasons vary. Most Qjump programs require the Extended Environment because it makes it simple to provide the type of pop-up menus and non-destructive windows that we prefer to use. Qtyp requires the Spell extensions, because we thought that it was necessary to separate out the actual spelling checking so that it could be used in other programs as well (such as real word processors). The Editor requires the Turbo Toolkit because it is Turbo compiled SBASIC and uses some facilities not available in SMSQ/E.

As a general rule, a BOOT file should load all the resident extensions you require, before any programs are started. This will avoid 'not complete' error messages when you try to load further extensions. The BOOT file is used in much commercial software to give users instant access to their new program. Many users never progress beyond this point, but re-boot their system every time they wish to change programs.

The boundary between a supplier providing a very complex BOOT file to make it very easy to use their software, and a supplier providing so complex a BOOT file that it becomes almost impossible to use any other supplier's software is a very fine one.

There is one simple test that you can do to find out whether a particular program is likely to give problems. Can you execute two copies of the program at the same time as the SBASIC interpreter? In the case of programs written for the Pointer Interface, the answer will usually be an unqualified yes. For other software you may have to do some detective work.

The manual for a software product should tell which of the files are resident extensions, and which are executable program files. If it does not, then you must first look at the BOOT file for the program. You need to find the command which **EXEC**s the program itself. Before this you may find some **RESPR**, **LBYTES**, **CALL** or **LRESPR** commands. If you find any **POKE**s, you can probably give up. Next, reset QPC and **LOAD** the BOOT file, delete all of the SBASIC commands except the **RESPR**, **LBYTES**, **CALL**, **LRESPR** and **EXEC** commands you have found and add a second **EXEC** for the program. Now **RUN** this skeleton SBASIC BOOT program - you should now be able to press CTRL and C to switch the keyboard from one copy of the program, to the next and then to the SBASIC interpreter. If it turns out that the program is so badly behaved that you cannot have two copies executing at the same time, then it is unlikely that the program will tolerate any other software.

If you cannot execute the second copy of the program because there is not enough memory left, then you will need to use the Psion option of the HOTKEY System 2.

If, while using a particular program, you find that bits of its windows tend to disappear, or get eaten up by other programs, then you will need to execute the program with a Guardian window using the HOTKEY System 2. If the program keeps on modifying the display while it is buried, then you will need a Guardian window with the Freeze option.

To set up your own BOOT file, you will have to determine which resident extensions are needed for each of the programs you wish to use. This should be stated in the manual. Alternatively, you can examine the supplier's own BOOT file. Resident procedures will be any code loaded by statements of the form:

**a=RESPR(size): LBYTES flpl_filename,a: CALL a** or
**LRESPR (filename)** or
**base=RESPR (size)** loading several files into one space
**LBYTES flpl_filenamel, base: CALL base LBYTES flpl_filename2, base + a_bit:**
**CALL base + a_bit** . . . etc.

These statements can be copied into your own BOOT file at the appropriate point, and the files themselves copied onto your own BOOT disk. The statements may be scattered over several lines to confuse you.

Sorting out BOOT files varies from the easy (e.g. The Editor) to the impossible (too many to mention). Very easy BOOT files would consist of "**EXEC flp1_filename**", in which case you need to add nothing to your own BOOT file unless you wish to HOTKEY the program with **HOT_RES**, **HOT_CHP** or **HOT_LOAD**. Difficult conversions are where the BOOT file indulges in copyright messages, pretty borders, playing tunes or other methods of obscuring the useful bits of code. Impossible BOOT files are those which include **POKE**s, or start an application with a **CALL** statement. These can, sometimes be used, but require the attention of an expert machine code hacker to convert them to a sanitary form.

Some resident extensions interact with others. If this happens, then some care is required with the ordering of the resident extensions. The HOTKEY System 2 interacts with the ALTKEY facility. The Pointer Interface interacts with Lightning.

Load any program specific resident extensions and other Toolkits. When all your resident extensions have been loaded, you should set up the Hotkeys you require, and include a **HOT_GO** command to get the HOTKEY Job going.

## Examples

These example BOOT files are intended to start you off. We specify the drive explicitly, and the file names are between apostrophes. The first is for clarity only, the second is a personal preference.

## BOOT_PSION Program

This boot file sets up a Psion plus Qtyp environment. All four Psion programs are permanently resident, although only Quill is started.

```
100 REMark - Load all our extensions
110 :
160 LRESPR ' f lpl_qtyp_spell'                  spelling checker extensions
170 :
180 REMark - Extensions loaded, stuff our QPC full of the
190 REMark - Psion programs
200 :
210 ERT HOT_RES ('t', ' f lpl_qtyp' )           Qtyp in case we use Quill
220 ERT HOT_RES ('q', 'f lpl_quill', p)         ALT Q for a new Quill
230 ERT HOT_RES ('a', 'f lpl_abacus', p)        ALT A for Abacus
240 ERT HOT_RES ('r',  'f lpl_archive', p)      ALT R for Archive
250 ERT HOT_RES ('e', 'f lpl_easel', p)         ALT E for a new Easel
260 :
270 HOT_GO                                      get HOTKEY going
280 :
290 : REMark - now we set some HOTKEYS for picking Jobs
300 : REMark - to pretend that we are using Taskmaster
310 :
320 ERT HOT_PICK (' 0' , '' )                   SBASIC and other no-name Jobs
330 ERT HOT_PICK ('1',  'Quill')
340 ERT HOT_PICK ('2',  'Abacus')
350 ERT HOT_PICK ('3',  'Archive')
360 ERT HOT_PICK ('4',  'Easel')
370 HOT_LIST                                    tell us what we have, please
380 PAUSE 300: HOT_DO q                         start with Quill only
```

## BOOT_ANOTHER Program

```
100 REMark - First shrink SBASIC' s windows a bit
110 WINDOW #0;512, 42, 0,214:BORDER #0;1,4,0
120 WINDOW #1;256,172,256,36:BORDER #1;1,255
130 WINDOW #2; 256,172, 0,36:BORDER #2;1,255
150 :
210 LRESPR 'flp1_qtyp_spell'          spelling checker extensions
220 LRESPR 'flp1_Qpac2'
230 :
240 ERT HOT_WAKE ('x',  'Exec')       Exec menu of QPAC 2
250 ERT HOT_WAKE ('p', 'Pick')        Pick menu of QPAC 2
260 ERT HOT_RES  ('t',  'flp1_qtyp')  Qtyp
270 ERT HOT_RES ('c', ' flp1_calc')   Pop up calculator
280 ERT HOT_RES ('k', ' flp1_calendar') ... our calendar
290 ERT HOT_RES ('w', ' flp1_alarm')  ... and the alarm
300 ERT HOT_LOAD ('d', ' flp1_QD')    load QD on demand
310 ERT HOT_LOAD ('8',  ' flp1_Text87') ...orText87
330 HOT_GO                            get HOTKEY going as well
340 EXEC ' f lp1_Clock'               clock around the clock
350 EXEC ' f lp1_Sysmon'              we need this to know what is going on
360 :
370 ERT HOT_PICK ('b', '')            pick SBASIC
380 HOT_LIST
390 PAUSE 300: HOT_DO e               start off with the Editor
400 PAUSE 100: HOT_DO b               but with SBASIC on top
```

## BOOT_HI_RES

BOOT program for a high resolution system

```
100 REMark Hi Res BOOT program
110 :
120 REMark For 800x600 display in 65536 colours
130 :
140 DISP_COLOUR 3,800,600
150 COLOUR_QL
160 BGCOLOUR_24 1.679652E7
170 WINDOW#0,800,120,0,330 : BORDER#0,1,1
180 WINDOW#1,400,300,400,30 : BORDER#1,1,255
190 WINDOW#2,400,300,0,30 : BORDER#2,1,255
200 INK#0,7 : PAPER#0,0 : CSIZE#0,1,1
210 INK#1,0 : PAPER#1,7
220 INK#2,7 : PAPER#2,1
230 CLS#0 : CLS#1 : CLS#2
240 :
250 REMark Load some resident extensions
260 LRESPR win1_system_qlib_run336mod     Qliberator runtimes
270 LRESPR win1_system_menu_rext          Menu extensions
320 :
330 REMark Set up the pointer system
340 LRESPR win1_pe_qpac_qpac2
350 :
390 REMark Set up Hot keys
400 ERT HOT_PICK('b','')                  Pick Basic
410 ERT HOT_THING('1','Files';'\dWIN1_')  Files thing for WIN1_
420 ERT HOT_THING(CHR$(232),'Button_sleep') Sleep thing
430 ERT HOT_THING('.','Button_Pick')      Button frame
440 ERT HOT_LOAD1('x','win1_xchange_xchange')
450 :
470 ERT HOT_WAKE('P','Pick')              Call the Pick menu
480 ERT HOT_WAKE('R','Rjob')              Call the Rjob menu
490 :
```

```
500 REMark Set up now go
510 :
520 REMark Create the buttons for the screen
530 :
550 BT_SLEEP 'Pick'
560 BT_SLEEP 'Exec'
570 BT_SLEEP 'Rjob'
580 BT_HOTKEY 'x','Xch'                    Put Xchange on a button
590 BT_HOTKEY '1','WIN1'                   Put Win1_ on a button
600 :
610 HOT_DO 'b'                             Pick the button frame
620 HOT_DO CHR$(232)                       Put system(SBASIC) to sleep
630 :
640 HOT_GO                                 Start the hotkey system
650 :
660 REMark SBASIC setups
670 PROG_USE win1_
680 DATA_USE win1_
```

## file types
## files

All I/O on QPC is to or from a 'logical file'. Various file types exist.

**data** SBASIC programs, text files. Created using **PRINT**, **SAVE**, accessed using **INPUT**, **INKEY$**, **LOAD** etc.

**exec** An executable transient program. Saved using **SEXEC**, loaded using **EXEC**, **EXEC_W** etc.

**code** Raw memory data, screen images, etc. Saved using **SBYTES**, loaded using **LBYTES**.

## flp
## floppy disk drive
## device

QPC can access 2 floppy disk drives (called FLP1_ and FLP2_) which correspond to the A: and B: drive of the host PC.

QPC will support double density (DD) and high density (HD) QDOS and MSDOS formatted disks .

Double density disks will store up to 720K bytes (1440 sectors) of data, and high density disks will store up to 1.4M bytes (2880 sectors) of data.

# functions and procedures

SBASIC functions and procedures are defined with the **DEFine FuNction** and **DEFine PROCedure** statements. A function is activated (or called) by typing its name at the appropriate point in a SBASIC expression. The function must be included in an expression because it is returning a value and the value must be used. A procedure is activated (or called) by typing its name as the first item in a SBASIC statement.

Data can be passed into a function or procedure by appending a list of **actual parameters** after the function or procedure name. This list is compared to a similar list appended after the name of the function or procedure when it was defined. This second list is called the **formal parameters** of the function or procedure. The formal parameters must be SBASIC variables. The actual parameters must be an array, an array slice or a SBASIC expression of which a single variable or constant is the simplest form.

Since the actual parameters are actual expressions, they must have an actual type associated with them. The formal parameters are merely used to indicate how the actual parameters must be processed and so have no type associated with them. The items in each list of parameters are paired off in order when the function or procedure is called and the formal parameters become equivalent to the actual parameters. There are three distinct ways of using parameters.

If the actual parameter is a single variable and if data is assigned to the formal parameter in the function or procedure then the data is also assigned to the corresponding actual parameter.

If the actual parameter is an expression then assigning data to the corresponding formal parameter will have no effect outside the procedure. Note that a variable can be turned into an expression by enclosing it within brackets.

If the actual parameter is a variable but has not previously been set then assigning data to the corresponding formal parameter will set the variable specified as the actual parameter.

Variables can be defined to be local to a function or procedure with the **LOCal** statement. Local variables have no effect on similarly named variables outside the function or procedure in which they are defined and so allow greater freedom in choosing sensible variable names without the risk of corrupting external variables. A local variable is available to any inside function or procedure called from the procedure function in which it is declared to be local unless the function or procedure called contains a further local declaration of the same variable name.

Functions and procedures in SBASIC can be used recursively. That is a function or procedure can call itself either directly or indirectly.

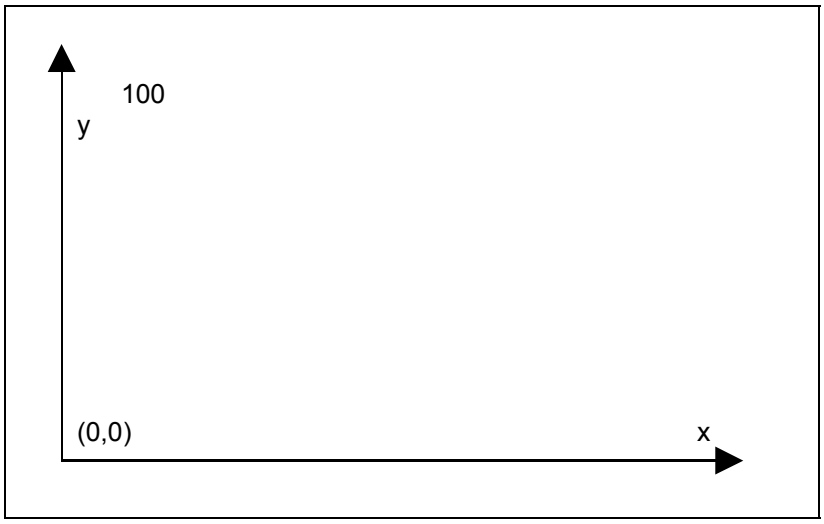
| Command | Function |
|---|---|
| **DEFine FuNction** | define a function |
| **DEFine PROCedure** | define a procedure |
| **RETurn** | leave a function or procedure (return data from a function) |
| **LOCal** | define local data in a function or procedure |

# graphics

It is important to realise that the QPC screen has non-square pixels and that changing screen mode will change the shape of the pixels. Thus if the graphics procedures were simply pixel based they would draw different shapes in the two modes. For example, in one mode we would have a circle while the same figure in the other mode would be an ellipse.

The graphics procedures ensure that whatever screen mode is in use, consistent figures are produced. It is not possible to use a simple pixel count to indicate sizes of figures, so instead the graphics procedures use an arbitrary scale and co-ordinate system to specify sizes and positions of figures.

The graphics procedures use the **graphics co-ordinate system**, i.e. draw relative to the **graphics origin** which is in the bottom left hand corner of the specified or default window. Note that this is not the same as the Pixel Origin used to define the position of Windows and Blocks etc. The graphics origin allows a standard Cartesian co-ordinate system to be used. A graphics cursor is updated after each graphics operation: subsequent operations can either be relative to this cursor or can be absolute, i.e. relative to the graphics origin.




The Graphics Coordinate System

The **scaling factor** is such that the full distance in the vertical direction in the specified or default window has length 100 by default and can be changed with the **SCALE** command. The scale in the x direction is equal to the scale in the y direction. However, the length of line which can be drawn in the x direction is dependent on the shape of the window. Increasing the scale factor increases the maximum size of the figure which can be drawn before the window size is exceeded. If the graphics output is switched to a different size of window then the subsequent size of the output is adjusted to fit the new window. If the figure exceeds its output window then the figure is clipped.

It is useful to consider the window to be a window onto a larger graphics space in which the figures are drawn. The **SCALE** command allows the graphics origin to be set so allowing the window to be moved around the graphics space.

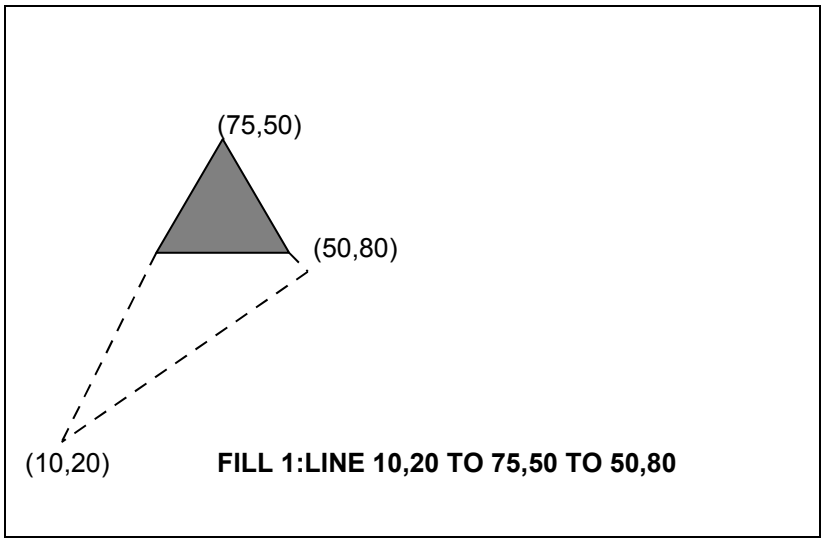
The graphics procedures are output to the window attached to the specified or default channel and the output is drawn in the **INK** colour for that channel.

| Command | Function | |
|---|---|---|
| **CIRCLE** | draw an ellipse or a circle | } |
| **LINE** | draw a line | } absolute |
| **ARC** | draw an arc of a circle | } |
| **POINT** | plot a point | } |
| **CIRCLE_R** | draw an ellipse or a circle | } |
| **LINE_R** | draw a line | } relative |
| **ARC_R** | draw an arc of a circle | } |
| **POINT_R** | plot a point | } |
| **SCALE** | set scale and move origin | |
| **FILL** | fill in a shape | |
| **CURSOR** | position text | |

**graphics fill**
Figures drawn with the graphics and turtle graphics procedures can be optionally 'filled' with a specified stipple or colour. If **FILL** is selected then the figure is filled as it is drawn.

The **FILL** algorithm stores a list of points to plot rather than actually plotting them. When the figure closes there are two points on the same horizontal line. These two points are connected by a line in the current ink colour and the process repeats. Fill must always be reselected before drawing a new figure to ensure that the buffer used to store the list of points is reset.

The following diagram illustrates **FILL**:




**warning:** There is an implementation restriction on **FILL**. **FILL** must not be used for re-entrant shapes (i.e. a shape which is concave). Re-entrant shapes must be split into smaller shapes which are not re-entrant and each sub-shape filled independently.

# history virtual device

The HISTORY virtual device is not associated with any physical hardware. HISTORY devices are buffers for storing information or passing it from one task to another. The HISTORY device is single ended, what goes in one end, comes out the same end in the reverse order (LIFO - last in first out).

A HISTORY device is much simpler than a PIPE device as it only has one end. It is used to store a number of messages which may then be retrieved in reverse order: if it becomes full, the oldest messages are thrown away. The messages are separated by newline characters.

There are two types of history devices: private and public. Private HISTORY devices are for use within a particular application and may only have one channel open to them. Public HISTORY devices are named and so may be accessed by many applications at the same time, or at different times. A public HISTORY device may even be used as a "mailbox".

A HISTORY device is opened by name, just like any other device. The name starts with "HISTORY" which is, for a public HISTORY device, followed by public name and then, optionally, the HISTORY device size. If no size is given, 1 kilobyte of message space is assumed. If a public HISTORY device already exists, then the size is ignored!

| | |
|---|---|
| **HISTORY** | A private HISTORY; 1024 bytes total space |
| **HISTORY _512** | A private HISTORY, 512 bytes total space |
| **HISTORY _thoughts** | A public HISTORY for thoughts |
| **HISTORY _box_80** | An 80 byte small mailbox called BOX |

Single character names should not be used: these are reserved as keys for special variations which may be made available in the future.

| | |
|---|---|
| **HISTORY _U_FILES** | A public HISTORY with all entries unique??? |

Messages may be put into a HISTORY device by either using **PUT** or **PRINT**. If the HISTORY device becomes full, the oldest message(s) are thrown away.

Messages may be taken out using **GET** or **INPUT**. But which message?

For a private HISTORY it is fairly simple. The first **GET** or **INPUT** after a message has been put into the HISTORY will get the most recent message. The next **GET** or **INPUT** will get the previous message until there are either no messages left (in which case **GET** or **INPUT** return null strings) or another message is put in. Note that GETting or INPUTing messages does not take them out of the HISTOTY.

| | |
|---|---|
| **OPEN_NEW #4, HISTORY _512** | Open a private HISTORY device to hold 512 bytes |
| **PRINT #4, msg1$** | For a private HISTORY, the message need not be atomic |
| **PUT #4, msg2$** | ...but this also puts a message in. |
| **INPUT #4, a$** | Inputs msg2$ into a$ |
| **GET #4,b$** | Gets msg1$ into b$ |

For a public HISTORY, the channels are fairly independent. A channel being used to read messages would continue to fetch messages in reverse order even if new messages are being added through other channels. In order to get the most recent message, a channel being used for read operations only needs to be able to reset its internal pointer. This is possible using the file positioning facility. Usually the position will be set to 0 (the most recent message) but it may be set to any (smallish) number.

| | |
|---|---|
| **GET #4\0, a$, b$** | Get the most recent and next most recent messages |
| **GET #4\4, x$** | Get the fifth most recent message. |

HISTORY has some characteristics of a filing system device. You can get a directory of public HISTORY devices, you can VIEW a public HISTORY and you can delete a public HISTORY.

| | |
|---|---|
| **DIR HISTORY** | Get a list of public HISTORY devices |
| **VIEW HISTORY _thoughts** | Have a look at my thoughts |
| **DELETE HISTORY _thoughts** | ...and get rid of them |

## identifier

An SBASIC identifier is a sequence of letters, numbers and underscores.

define: *letter* := | **a**..**z**
| **A**..**Z**

*number* := | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **0** |

*identifier* := *letter* * |[ *letter* | *number* | _ | ] *

example:
i. **a**
ii. **limit_1**
iii. **current_guess**
iv. **counter**

An identifier must begin with a letter followed by a sequence of letters, numbers and underscores and can be up to 255 characters long. Upper and lower case characters are equivalent.

Identifiers are used in the SBASIC system to identify *variables*, *procedures*, *functions*, *repetition* loops, etc.

**waring:** NO meaning can be attributed to an identifier other than its ability to identify constructs to SBASIC. SBASIC cannot infer the intended use of an identifier from the identifier's name!

## keyword

SBASIC keywords are identifiers which are defined in the SBASIC *Keyword Reference Guide*. Keywords have the same form as a SBASIC standard *identifier*. The case of the keyword is not significant. Keywords are echoed as a mixture of upper and lower case letters and are always reproduced in full. The upper case portion indicates the minimum required to be typed in for SBASIC to recognise the keyword.

The set of SBASIC keywords may be extended by adding *procedures* to QPC. It is a good idea to define these with their names in upper case and this will indicate their special function in the SBASIC system. Conversely, ordinary procedures should be defined with their names in lower case.

**warning:** Existing keywords cannot be used as ordinary identifiers within a SBASIC program. SBASIC keywords may be found by typing **EXTRAS** at the command prompt.

# maths
# functions

SBASIC has the standard trigonometrical and mathematical functions.

| Function | Name |
|---|---|
| **COS** | cosine |
| **SIN** | sin |
| **TAN** | tangent |
| **ATAN** | arctangent |
| **ACOT** | arcotangent |
| **ACOS** | arcosine |
| **ASIN** | arcsine |
| **COT** | cotangent |
| **EXP** | exponential |
| **LN** | natural logarithm |
| **LOG10** | common logarithm |
| **INT** | integer |
| **ABS** | absolute value |
| **RAD** | convert to radians |
| **DEG** | convert to degrees |
| **PI** | return the value of $\pi$ |
| **RND** | generate a random number |
| **RANDOMISE** | reseed the random number generator |

## mdv
## microdrive directory device

Microdrives provided the main permanent storage on the Sinclair QL. Each Microdrive cartridge had a capacity of at least 100Kbytes.

Microdrives are not supported in QPC.

## nul virtual
## device

The NUL virtual device is not associated with any physical hardware. NUL devices are completely dummy.

The NUL device may be used in place of a real device. The NUL device is usually used to throwaway unwanted output. It may, however, be used to provide dummy input or to force a job to wait forever. There are five variations.

**NULP** waits (forever or until the specified timeout) on any input or output operation.

**NUL**, **NULF**, **NULZ** and **NULL** ignore all operations (the output is thrown away).

**NUL**, **NULF**, **NULZ** and **NULL** return a zero size window in response to window information requests. Pointer Information calls (IOP.PINF , IOP.RPTR) return an invalid parameter error.

**NUL** is an output only device, all input operations return an invalid parameter error.

**NULF** emulates a null file. Any attempt to read data from NULF will return an End of File Error as will any file positioning operation. Reading the file header will return 14 bytes of zero (no length, no type).

**NULZ** emulates a file filled with zeros. The file position can be set to anywhere. Reading the file header will return 14 bytes of zero (no length, no type).

**NULL** emulates a file filled with null lines. The file appears to be full of the newline character (10). The file position may be set to anywhere. Reading the file header will return 14 bytes of zero (no length, no type).

## operators

| Operator | Type | Function |
|---|---|---|
| **=** | floating string | logical type 2 comparison |
| **==** | numeric string | almost equal ** (type 3 comparison) |
| **+** | numeric | addition |
| **-** | numeric | subtraction |
| **/** | numeric | division |
| ***** | numeric | multiplication |
| **<** | numeric string | less than (type 2 comparison) |
| **>** | numeric string | greater than (type 2 comparison) |
| **<=** | numeric string | less than or equal to (type 2 comparison) |
| **>=** | numeric string | greater than or equal (type 2 comparison) |
| **<>** | numeric string | not equal to (type 3 comparison) |
| **&** | string | concatenation |
| **&&** | bitwise | AND |
| **\|\|** | bitwise | OR |
| **^^** | bitwise | XOR |
| **~** | bitwise | NOT |
| **OR** | logical | OR |
| **AND** | logical | AND |
| **XOR** | logical | XOR |
| **NOT** | logical | NOT |
| **MOD** | integer | modulus |
| **DIV** | integer | divide |
| **INSTR** | string | type 1 string comparison |
| **^** | floating | raise to the power |
| **-** | floating | unary minus |
| **+** | floating | unary plus |

**almost equal - equal to 1 part in $10^7$

If the specified logical operation is true then a value not equal to zero will be returned. If the operation is false then a value of zero will be returned.

**precedence** The precedence of SBASIC operators is defined in the table above. If the order of evaluation in an expression cannot be deduced from this table then the relevant operations are performed from left to right. The inbuilt precedence of SBASIC operators can be overriden by enclosing the relevant sections of the expression in parentheses.

*highest* unary plus and minus
string concatenation
INSTR
exponentiation
multiply, divide, modulus and integer divide
add and subtract
logical comparison
NOT (bitwise or logical)
AND (bitwise or logical)
*lowest* OR and XOR (bitwise or logical)

# pipe
# virtual device

The PIPE virtual device is not associated with any physical hardware. PIPE devices are buffers for storing information or passing it from one task to another. The PIPE is double ended: what goes in one end, comes out the other in the same order (FIFO - first in first out).

There are two variations on the PIPE driver: named and unnamed pipes. Both of these are used to pass data from one program to another. Unnamed pipes cannot be opened with the SBASIC **OPEN** commands but are opened automatically by the **EX** and **EW** commands when these are required to set up a "production line" of Jobs. Whereas, if a pipe is identified by a name, any number of Jobs (including SBASIC) can open channels to it as either inputs or outputs.

If, using named pipes, matters become confused, then that is a problem to be solved by the Jobs themselves. This is not as bad as it sounds. Unlike other devices, named pipes transfer multiple byte strings atomically unless the pipe allocated is too short to hold the messages. This means that provided the messages are shorter than the pipe, many jobs can put messages into a named pipe and many jobs can take messages out of a named pipe without the messages themselves becoming scrambled.

If a PIPE is shared in this way, there are two simple ways of ensuring that the messages are atomic. The first, using fixed length messages, if not available to SBASIC programs. The second, using "lines" terminated by the newline character, works perfectly. N.B. the standard **PRINT** command will not necessarily send a line as a single string for each item output.

| | |
|---|---|
| **PRINT #3,a$ \ b$** | Bad, sends 4 strings: the newline characters are separate |
| **PRINT #3,a$ & CHR$ {10);** | Good, sends 1 string, including the newline |
| **INPUT #4,b$** | Good, reads a single line from the pipe |

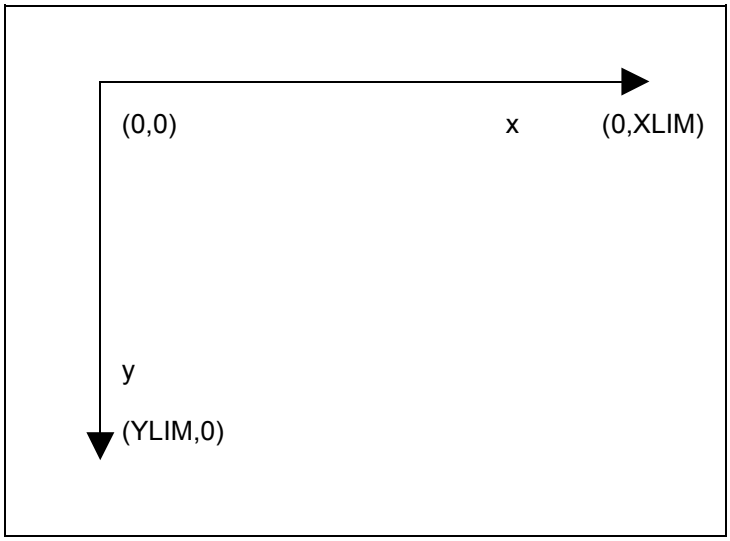
Named pipes should be opened with **OPEN_NEW** (**FOP _NEW**) for output and **OPEN_IN** (**FOP _IN**) for input. A named pipe is created when there is an open call for a named pipe, which does not exist. It goes away when there are no longer any channels open to it and it is has been emptied. As well as the name, it is possible to specify a length for a named pipe. If the pipe already exists, the length requested is ignored.

| | |
|---|---|
| **OPEN_NEW #4, PIPE_xp1** | Open named output pipe of default length (1024 bytes) |
| **OPEN_NEW #5, PIPE_frd_2048** | Open named output pipe of length 2048 bytes |
| **OPEN_IN #6, PIPE_xfr** | Open named input pipe |

# pixel coordinate system

The **pixel coordinate system** is used to define the positions and sizes of windows, blocks and cursor positions on the QPC screen. The coordinate system has its origin in the top left hand corner of the default window (or screen). The system will use the nearest pixel available for the particular mode set making the coordinate system independent of the screen mode in use.

Some commands are always relative to the default window origin, e.g. **WINDOW**, while some are always relative to the current window origin, e.g. **BLOCK**




The Pixel Coordinate System

## program

An SBASIC program consists of a sequence of SBASIC statements, where each statement is preceded by a line number. Line numbers are in the range of 1 to 32767.

| Command | Function |
|---|---|
| **RUN** | start a loaded program |
| **LRUN** | load a program from a device and start it |
| [**CTRL**] [**SPACE**] | force a program to stop |

syntax: *line_number* := *[*digit*]* {range 1..32767}

*[*line_number statement* *[:*statement*]*]*

example: i.
```
100 PRINT "This is a valid line number"
RUN
```

ii.
```
100 REMark a small program
110 COLOUR_QL
120 FOR foreground = 0 TO 7
130   FOR contrast = 0 TO 7
140     FOR stipple = 0 TO 3
150       PAPER foreground, contrast, stipple
160       CURSOR 0,70
170       FOR n = 0 TO 2
180         SCROLL 2,1
190         SCROLL -2,2
200       END FOR n
210     END FOR stipple
220   END FOR contrast
230 END FOR foreground
RUN
```

## QDOS
## QL

The QDOS operating system is a predecessor of the SMSQ/E operating system. QDOS was originally used in the Sinclair QL computer. Circa 1983

The Sinclair QL used a version of BASIC known as SuperBASIC. SBASIC and SMSQ/E used in QPC are direct descendants of SuperBASIC, and QDOS.

SMSQ/E includes all the QL SuperBASIC commands, the Toolkit 2 commands, Pointer interface, Window manager, and the commands which have been provided to support the various add-on drivers. SMSQ/E supports 99.9% of SuperBASIC. SMSQ/E supports all the devices, which were supported by the drivers supplied with the Atari QL Emulator, the GOLD card and the QXL. The (Super) Gold card is a hardware add-on to the Sinclair QL, and the QXL is a PC add-on card containing a QL compatible computer.

# ram
# virtual directory device

The RAM device behaves like a very fast disk drive. It is so fast because being virtual, there is virtually nothing to move to get information in and out. It is, in fact, no more than a reserved area of SMSQ/E's main memory (its RAM - Random Access Memory). This means, of course, that any space taken by a RAM disk is not available to programs executing in QPC. Furthermore, any data stored in a RAM disk will be lost when QPC is exited or reset!

RAM disks in the QPC may be of any size, subject to there being enough memory. The normal usage of a RAM disk would be to use it as a temporary storage area, or to speed up the operation of programs with very disk intensive operations.

A dynamic RAM Disk is created just by accessing it with any normal operation (e.g. DIR). This type of RAM Disk takes memory as required, and releases any memory as files are deleted or truncated.

A fixed RAM disk is created by formatting it: the size, in sectors, is given in place of the usual medium name. This pre-allocates all the space that will be available in the RAM disk.

**FORMAT ram2_80**

This removes the old RAM disk number 2, and sets up a new RAM disk of 80 sectors. A RAM disk may be removed by giving either a null name or zero sectors

**FORMAT ram1_** or **FORMAT ram1_0**

The RAM disk number should be between 1 and eight, inclusive, while the number of sectors (512 bytes) is only limited by the memory available.

# SMSQ/E

SMSQ/E is the QPC Operating System used by QPC, and supervises:

Task Scheduling and resource allocation
Screen I/O (including windowing)
Disk drive I/O
Parallel and serial channel communication
Keyboard input
Memory management

A full description of SMSQ is beyond the scope of this guide but a brief description is included.

**system calls**

System calls are processed by SMSQ in **supervisor mode**. When in supervisor mode, SMSQ will not allow any other job to take over the processor. System calls processed in this way are said to be **atomic**, i.e. the system call will process to completion before relinquishing the processor. Some system calls are only **partially atomic**, i.e. once they have completed their primary function they will relinquish the processor if necessary. Unless specifically requested all the system calls are partially atomic.

The standard mechanism for making a system call is by making a trap to one of the SMSQ system vectors with appropriate parameters in the processor registers. The action taken by SMSQ following a system call is dependent on the particular call and the overall state of the system at the time the call was made.

**input/output**

SMSQ supports a multitasking environment and therefore a file can be accessed by more than one process at a time. The SMSQ filing sub-system can handle files which have been opened as **exclusive** files or as **shared** files. A shared file cannot be written to. QPC devices are processed by the **serial I/O sun-system**. The filing sub-system and the serial I/O sub-system together make up the **redirectable I/O system**. As its name suggests any data output by this system can be redirected to any other device also supported by the redirectable I/O system.

The device names required by SMSQ are the same as the device names required by SBASIC and are discussed in the concept section *devices*. The collection of standard devices supplied with QPC can be expanded.

**devices**
The standard devices included in the system are discussed in this guide in the section **devices**. Further devices may be added to the system, given a name (e.g. SER1, PRT) and then accessed in the same way as any other QPC device.

**multitasking**
Jobs will be allowed a share of the CPU in line with their priority and competition with other jobs in the system. Jobs running under the control of SMSQ can be in one of three states:

**active:** Capable of running and sharing system resources. A job in this state may not be running continuously but will obtain a share of the CPU in line with its priority.

**suspended:** The job is capable of running but is waiting for another job or I/O. A job may be suspended indefinitely or for a specific period of time.

**inactive:** The job is incapable of running, its priority is 0 and so it can never obtain a share of the CPU

SMSQ will reschedule the system automatically at a rate related to the 50 Hz frame rate. The system will also be rescheduled after certain system calls.

example: This program generates an on-screen readout of the real-time clock, running as an independent job.

First **RUN** this program with a formatted disk in floppy drive 1. This generates a machine code title called 'clock'. Wait for the drive to stop.

Then type:

```
EXEC flp1_clock
```

and a continuous time display will appear at the top right of the command window.

```
100  c=RESPR(100)
110  FOR i = 0 TO 68 STEP 2
120    READ x:POKE_W i+c,x
130  END FOR i
140  SEXEC flp1_clock,c,100,256
1000 DATA 29439,29697,28683,20033,17402
1010 DATA 48,13944,200,20115,12040
1020 DATA 28691,20033,17402,74,-27698
1030 DATA 13944,236,20115,8279,-11314
1040 DATA 13944,208,20115,16961,16962
1050 DATA 30463,28688,20035,24794
1060 DATA 0,7,240,10,272,200
```

N.B. Line 1060 governs the position and colour of the clock window - the data terms are, in order:

border colour/width, paper/ink colour, window width, height, x-origin, y-origin

These are pairs of bytes, entered by **POKE_W** as words.

The x-origin and the y-origin (the last data item) should be 272 and 202 in monitor mode.

Generate the paper and ink word, for example, as 256*paper+ink. Thus white paper, red ink is 256*7 + 2 = 1794

## repetition

Repetition in SBASIC is controlled by two basic program constructs. Only the **FOR** construct must be identified to SBASIC:

```
REPeat [identifier]          FOR identifier = range
   statements                   statements
END REPeat [identifier]      END FOR [identifier]
```

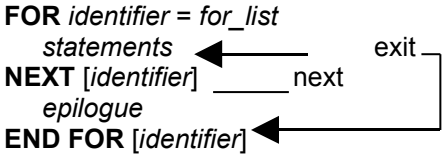
These two constructs are used in conjunction with two other SBASIC statements:

```
NEXT [identifier]            EXIT [identifier]
```

Processing a **NEXT** statement will either pass control to the statement following the appropriate **FOR** or **REPeat** statement, or if a **FOR** range has been exhausted, to the statement following the **NEXT**.

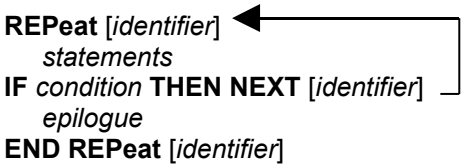
Processing an **EXIT** will pass control to the statement after the **END FOR** or **END REPeat** selected by the **EXIT** statement. **EXIT** can be used to exit through many levels of nested repeat structures. **EXIT** should always be used in **REPeat** loops to terminate the loop on some condition.

A combination of **NEXT**, **EXIT** and **END** statements allows **FOR** and **REPeat** loops to have a **loop epilogue** added. A loop epilogue is a series of SBASIC statements which are executed on some special condition arising within the loop:




The loop epilogue is only processed if the **FOR** loop terminates normally. If the loop terminates via an **EXIT** statement then processing will continue at the **END FOR** and the epilogue will not be processed.

It is possible to have a similar construction in a **REPeat** loop:




This time entry into the loop epilogue is controlled by the **IF** statement. The epilogue will or will not be processed depending on the condition in the **IF** statement. A **SELect** statement can also be used to control entry into the epilogue.

## screen

Some QL programs write directly to the display memory. This can cause display problems with QPC, as the display memory is not in the same place as in the QL.

The **QPC_QLSCREMU** command enables or disables the original QL screen emulation. When emulating the original screen, all memory write accesses to the area $20000-$207FFF are intercepted and translated into writes to the first 512x256 pixels of the big screen area. If the screen is in high colour mode, additional colour conversion is done.

Possible values are:

-1: automatic mode
0: disabled (default)
4: force to 4 colour mode
8: force to 8 colour mode

When in QL colour mode the emulation just transfers the written bytes to the larger screen memory, i.e. when the big mode is in 4 colour mode, the original screen area is also treated as 4 colour mode. In high colour mode however the colour conversion can do both modes. In this case you can pre-select the emulated mode (4, 8 as parameter) or let the last issued **MODE** call decide (automatic mode). Please note that that the automatic mode does not work on a per-job basis, so any job which issues a **MODE** command changes the behaviour globally.

Please also note that this transition is one-way only, i.e. bytes written legally to the first 512x256 pixels are not transferred back to the original QL screen (in case of a high colours screen this would hardly be possible anyway). Unfortunately this also means that not all old programs run perfectly with this type of emulation. If you experience problems start the misbehaving application in 512x256 mode.

# slicing

Under certain circumstances it is possible to refer to more than one element in an array i.e. slice the array The array slice can be thought of as defining a **subarray** or a series of subarrays to SBASIC. Each slice can define a continuous sequence of elements belonging to a particular dimension of the original array. The term array in this context can include a numeric array, a string array or a simple string.

It is not necessary to specify an index for the full number of dimensions of an array. If a dimension is omitted then slices are added which will select the full range of elements for that particular dimension, i.e. the slice (0 TO ). SBASIC can only add slices to the end of a list of array indices.

syntax: *index* := | *numeric_exp* {single element}
| *numeric_exp* **TO** *numeric_exp* {range of elements}
| *numeric_exp* **TO** {range to end}
| **TO** *numeric_expression* {range from beginning}

*array_reference* := | *variable*
| *variable* **(** [ *index* * [,*index*] * ] **)**

An array slice can be used to specify a source or a destination subarray for an assignment statement.

example: i. **PRINT data_array**
ii. **PRINT letters$(1 TO 15)**
iii. **PRINT two_d_array (3 , 2 TO 4)**

String slicing is performed in the same way as slicing numeric or string arrays.

Thus

| | |
|---|---|
| **a$(n)** | will select the nth character. |
| **a$(n TO m)** | will select all characters from the nth to the mth, inclusively |
| **a$(n TO)** | will select from a character n to the end, inclusively |
| **a$(1 TO m)** | will select from the beginning to the nth character inclusively |
| **a$** | will select the entire contents of a a$ |

Some forms of BASIC have functions called **LEFT$**, **MID$**, **RIGHT$**. These are not necessary in SBASIC. Their equivalents are specified below:

| SBASIC | Other BASIC |
|---|---|
| **a$(n)** | **MID$(a$,n,1)** |
| **a$(n TO m)** | **MID$ (a$,n,m+1-n)** |
| **a$(1 TO n)** | **LEFT$ (a$,n)** |
| **a$(n TO)** | **RIGHT$ (a$,LEN(a$)+1-n)** |

**warning:** Assigning data to a sliced string array or string variable may not have the desired effect. Assignments made in this way will not update the length of the string. The length of a string array or string variable is only updated when an assignment is made to the whole string.

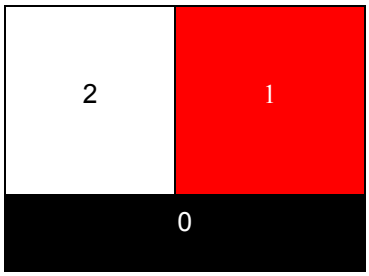
## start up

Immediately after starting or resetting QPC the screen will be cleared and the default screen is displayed.

QPC has the ability to 'boot' itself up from programs contained in either the Hard disk or in Floppy drive 1. If a disk is found and if it contains a file called **BOOT** it is loaded and run.

**default screen**

The QL has three default channels which are linked to three default windows.




Channel 0 is used for listing commands and error messages, channel 1 for program and graphics output and channel 2 for program listings. The default channel can be modified using the optional channel specifier in the relevant command.

# sound

QPC can emit sound by playing sampled data using the SMSQ/E Samples Sound System (not explained here) or by an emulation of the QL's second processor (an 8049). The later is controlled by specifying:

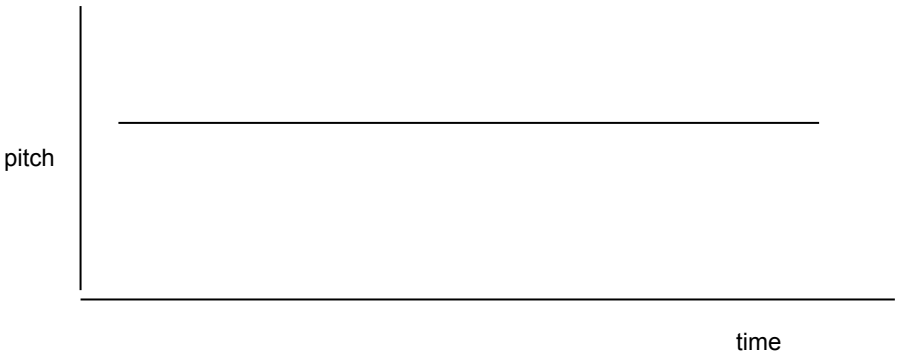
up to two pitches
the rate at which the sound must move between the pitches, the ramp
how the sound is to behave after it has reached one of the specified pitches, the wrap
if any randomness should be built into the sound, i.e. deviations from the ramp
if any fuzziness should be built into the sound. i.e. deviations on every cycle of the sound

Fuzziness tends to result in buzzy sounds while randomness, depending on the other parameters, will result in 'melodic' sounds or noise.

The complexity of the sound can be built up stage by stage gradually building more complex sounds. This is, in fact, the best way to master sound on QPC.

Specify a duration and a single pitch. The specified pitch will be beeped for the specified time.

**LEVEL 1**

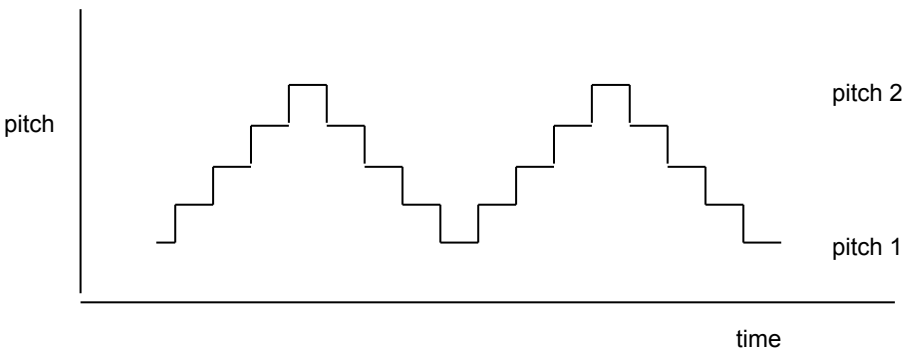



This is the simplest sound command, other than the command to stop the sound, on QPC.

**LEVEL 2**

A second pitch and a gradient can be added to the command. The sound will then 'bounce' between the two pitches at the rate specified by the gradient.

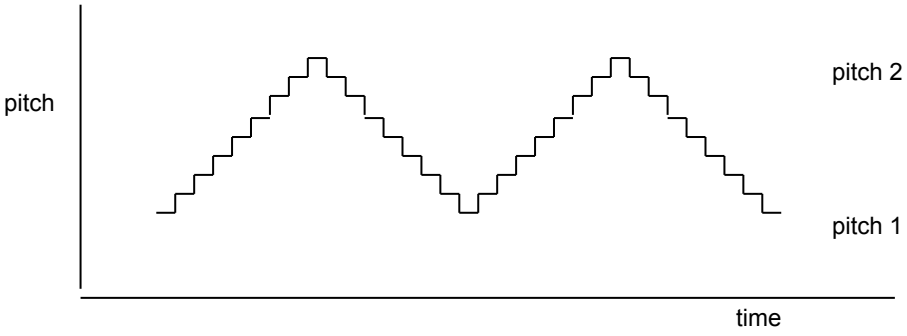
The sounds produced at this level can vary between: semi musical beeps, growls, zaps and moans. It is best to experiment.

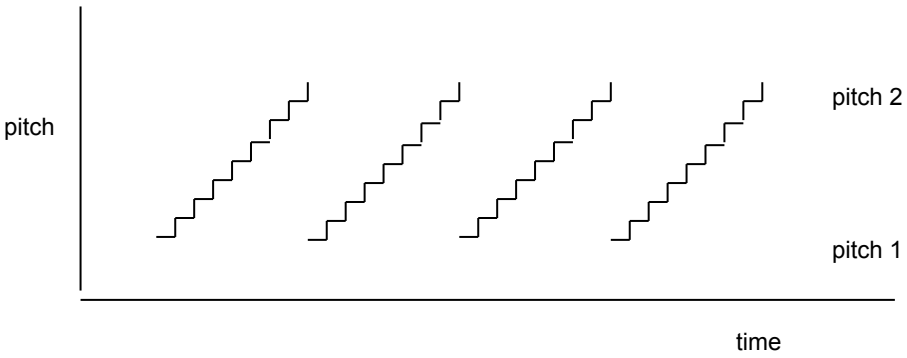


**LEVEL3**

A parameter can be added which controls how the sound behaves when it becomes equal to one of the specified pitches. The sound can be made to 'bounce' or 'wrap'.

The number of wraps can be specified, including wrap forever. It is even more important to experiment.

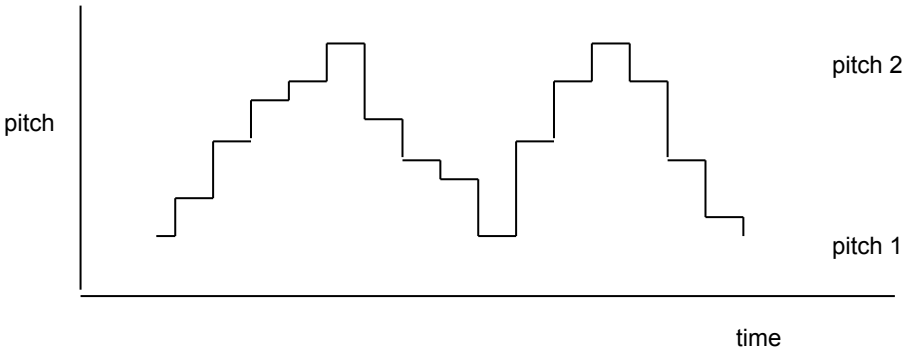



**LEVEL4**

Randomness can be added to the sound. This is a deviation from the specified step or gradient.

Depending on the amount of randomness added in relation to the pitches and the gradient, it will generate a very wide and unexpected range of sounds.



**LEVEL 5**

More variation can be added by specifying 'fuzziness'. Fuzziness adds a random factor to the pitch continuously Fuzziness tends to make the sound buzz.

Combining all of the above effects can make a very wide range of sounds, many of them unexpected. QPC sound is best explored through experiment. By specifying a time interval of zero the sound can be made to repeat forever and so a sequence of **BEEP** commands can be used until the sound generated is the sound which is required. A word of warning: slight changes in the value of a single parameter can have alarming results on the sound generated.

## statement

An SBASIC statement is an instruction to QPC to perform a specific operation, for example:

**LET a = 2**

will assign the value 2 to the variable identified by **a**.

More than one statement can be written on a single line by separating the individual statements from each other by a colon ( : ), for example:

**LET a = a + 2 : PRINT a**

will add 2 to the value identified by the variable **a** and will store the result back in **a**. The answer will then be printed out

If a line is not preceded by a line number then the line is a direct command and SBASIC processes the statement immediately. If the statement is preceded by a line number then the statement becomes part of a SBASIC program and is added into the SBASIC program area for later execution.

Certain SBASIC statements can have an effect on the other statements over the rest of the logical line in which they appear i.e. **IF**, **FOR**, **REPeat**, **REM**, etc. It is meaningless to use certain SBASIC statements as direct commands.

## string arrays
## string variables

String arrays and numeric arrays are essentially the same, however there are slight differences in treatment by SBASIC. The last dimension of a string array defines the maximum length of the strings within the array. String variables can be any length up to 32766. Both string arrays and string variables can be sliced.

String lengths on either side of a string assignment need not be equal. If the sizes are not the same then either the right hand string is truncated to fit or the length of the left hand string is reduced to match. If an assignment is made to a sliced string then if necessary the 'hole' defined by the slice will be padded with spaces.

It is not necessary to specify the final dimension of a string array. Not specifying the dimension selects the whole string while specifying a single element will pick out a single character and specifying a slice will define a sub string.

comment: Unlike many BASICs SBASIC does not treat string arrays as fixed length strings. If the data stored in a string array is less than the maximum size of the string array then the length of the string is reduced.

**warning:** Assigning data to a sliced string array Or string variable may not have the desired effect. Assignments made in this way will not update the length of the string and so it is possible that the system will not recognise the assignment. The length of a string array or a string variable is only updated when an assignment is made to the whole string.

| Command | Function |
|---|---|
| **FILL$** | generate a string |
| **LEN** | find the length of a string |

# string comparison

**order** . (decimal point/full stop)
digits or numbers in numerical order
**AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz**
space **! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ | ] ^ _ / { | } ~ ©**
other non printing characters

The relationship of one string to another may be:

equal: All characters or numbers are the same or equivalent

lesser: The first part of the string, which is different from the corresponding character in the second string, is before it in the defined order.

greater: The first part of the first string which is different from the corresponding character in the second string, is after it in the defined order.

Note that a '.' may be treated as a decimal point in the case of string comparison which sorts numbers (such as SBASIC comparisons). Note also that comparison of strings containing non-printable characters may give unexpected results.

**types of comparison**

type 0 case dependent - character by character comparison

type 1 case independent - character by character

type 2 case dependent - numbers are sorted in numerical order

type 3 case independent - numbers are sorted in numerical order

type 0 not normally used by the SBASIC system.

**Usage** type 1 File and variable comparison
type 2 SBASIC <, <=, =, >= ,>, **INSTR** and <>
type 3 SBASIC == (equivalence)

# syntax definitions

SBASIC syntax is defined using a non-rigorous 'meta language' type notation. Four types of construction are used :

| | |
|---|---|
| \| \| | Select one of |
| [ ] | Enclosed item(s) are optional |
| * * | Enclosed items are repeated |
| .. | Range |
| { } | Comment |

e.g.

| | |
|---|---|
| \| A \| B \| | A or B |
| [ A ] | A is optional |
| * A * | A is repeated |
| A..Z | A, B, C, etc |
| {this is a comment} | |

Consider a SBASIC identifier.

A sequence of numbers, digits, underscores, starting with a letter and finishing with an optional % or $

*letter* := | **A**..**Z**
| **a**..**z**
{a letter is one of: ABCDEFGHIJKLMNOPQRSTUVWXYZ}
or abcdefghijklmnopqrstuvwxyz

*digit* := | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |

{a digit is 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9}

*underscore* := _
{an underscore is _ }

*identifier* := *letter* * [*letter* | *digit* | *underscore* ] * | % | $ |

| | |
|---|---|
| ------- | ---------- |
| must start with a letter | a sequence of letters digits and underscores i.e. repeat something which is optional |

## turtle graphics

SBASIC has a set of turtle graphics commands:

| Command | Function |
|---|---|
| **PENUP** | stop drawing |
| **PENDOWN** | start drawing |
| **MOVE** | move the turtle |
| **TURN** | turn the turtle |
| **TURNTO** | turn to a specific heading |

The set of commands is the minimum and normally would be used within another procedure to expand on the commands. For example:

```
100 DEFine PROCedure forward(distance)
110    MOVE distance
120 END DEFine
130 DEFine PROCedure backwards(distance)
140    MOVE -distance
150 END DEFine
160 DEFine PROCedure left(angle)
170    TURN angle
180 END DEFine
190 DEFine PROCedure right(angle)
200    TURN -angle
210 END DEFine
```

These will define some of the more famous turtle graphic commands.

Initially the turtle's pen is up and the turtle is pointing at $0^0$, which is to the right hand side of the window.

The **FILL** command will also work with figures drawn with turtle graphics. Also ordinary graphics and turtle graphics can be mixed, although the direction of the turtle is not modified by the ordinary graphics commands.

## win hard disk directory device

Hard disk drives on QPC are large files stored on the host PC. The files usually have the suffix ".WIN" but anything else is fine, too. The name and directory can be configured separately for up to 8 win devices in the QPC program configurator.

# windows

Windows are areas of the screen which behave, in most respects, as though each individual window was a screen in its own right, i.e. the window will scroll when it has become filled by text, it can be cleared with the **CLS** command, etc.

Windows can be specified and linked to a channel when the channel is opened. The current window shape can be changed with the **WINDOW** command and a border added to a window with the **BORDER** command. Output can be directed to a window by printing to the relevant channel. Input can be directed to have come from a particular window by inputting from the relevant channel If more than one channel is ready for input then input can be switched between the ready channels by pressing

**[CTRL] C**

The cursor will flash in the selected window

Windows can be used for graphics and non-graphic output at the same time. The non graphic output is relative to the current cursor position which can be positioned anywhere within the specified window with the **CURSOR** command and at any line-column boundary with the **AT** command. The graphics output is relative to a graphics cursor which can be positioned and manipulated with the graphics procedures.

**parts**

Certain commands (**CLS**, **PAN** etc.) will accept an optional parameter to define part of the current window for their operation. This parameter is as defined below:

| part | description |
|---|---|
| 0 | whole screen |
| 1 | above and excluding cursor line |
| 2 | bottom of screen excluding cursor line |
| 3 | whole of cursor line |
| 4 | line right of and including cursor |

| Command | Function |
|---|---|
| **WINDOW** | re-define a window |
| **BORDER** | take a border from a window |
| **PAPER** | define the paper colour for a window |
| **INK** | define the ink colour for a window |
| **STRIP** | define a strip colour for a window |
| **PAN** | pan a window's contents |
| **SCROLL** | scroll a window's contents |
| **AT** | position the print position |
| **CLS** | clear a window |
| **CSIZE** | set character size |
| **FLASH** | character flash |
| **RECOL** | recolour a window |

## T



## U



## V



## W