Minerva4Q68
Concepts

The Concepts Reference Guide describes concepts relating to Minerva SuperBASIC and the Q68. It is best to think of the Concept Guide as a source of information. If there are any questions about SuperBASIC or the Q68 itself which arise out of using the computer 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 Minerva manual, and the Q68 manual.

The Minerva operating system was originally designed as a replacement ROM operating system for the Sinclair QL computer, currently licenced under GPLv3. This port is aimed at the Q68, an FPGA-based replacement board for the QL. It is not intended as a replacement for the SMSQ/E OS supplied with the Q68, as SMSQ/E is far more extensive and better suited to support the Q68 hardware than the 48K ROM-based Minerva. We just provide this port to demonstrate the Q68's ability to run 'old school' ROM images, give Q68 users the Minerva look and feel, and maybe provide an opportunity to run badly written software that doesn't run on SMSQ/E (but chances are big that this software won't run on Minerva either).

The current Minerva build is based on v1.98, with a few modifications to run successfully on the Q68.

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

BASIC
SuperBASIC includes most of the functions, procedures and constructs found in other dialects of BASIC. Many of these functions are superfluous in SuperBASIC 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 SuperBASIC 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 SuperBASIC
STR$	not required in SuperBASIC
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 SuperBASIC 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 SuperBASIC 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 **CTRL F5**.

To perform a soft reset,

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

See the **Keyboard Changes** section for more keystroke selections.

channels

A channel is a means by which data can be output to or input from a Q68 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 the Q68 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 Q68's 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
```

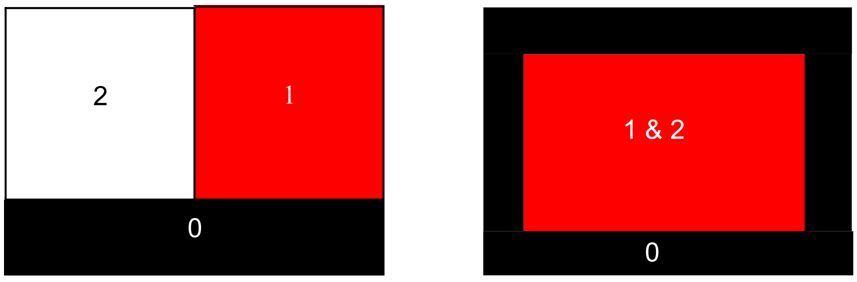
will output the text "this text is in file test_file" to the file test_file. It is important to close the file after all the accesses have been completed to ensure that all the data is written.

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$**.

A channel can be opened as a console channel; output is directed to a specified window on the Q68's screen and input is taken from the Q68's 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.

Minerva has three default channels which are opened automatically. Each of these channels is linked to a window on the Q68's screen.

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



Monitor Television

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

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.

The following table is for UK keyboards. Keyings may change with other nationalities

Decimal	Hex	Keying	Display/Function
0	00	CTRL £	$^{N}{}_{UL}$ / NULL
1	01	CTRL A	$^{F}{}_{1}$
2	02	CTRL B	$^{F}{}_{2}$
3	03	CTRL C	$^{F}{}_{3}$ / Change input channel (see note)
4	04	CTRL D	$^{F}{}_{4}$
5	05	CTRL E	$^{F}{}_{5}$
6	06	CTRL F	$^{A}{}_{K}$
7	07	CTRL G	♪
8	08	CTRL H	$^{B}{}_{S}$
9	09	TAB (CTRL I)	$^{H}{}_{T}$ / Next field
10	0A	ENTER (CTRL J)	New line / Command entry
11	0B	CTRL K	$^{V}{}_{T}$
12	0C	CTRL L	$^{F}{}_{F}$
13	0D	CTRL M	$^{C}{}_{R}$ / Enter
14	0E	CTRL N	$^{S}{}_{0}$
15	0F	CTRL O	$^{S}{}_{I}$
16	10	CTRL P	^{0}
17	11	CTRL Q	^{1}
18	12	CTRL R	^{2}
19	13	CTRL S	^{3}
20	14	CTRL T	^{4}
21	15	CTRL U	^{5}
22	16	CTRL V	^{6}
23	17	CTRL W	^{7}
24	18	CTRL X	^{8}
25	19	CTRL Y	^{9}
26	1A	CTRL Z	^{A}
27	1B	ESC (CTRL SHIFT |)	^{B} / Abort current level of command
28	1C		^{C}
29	1D	CTRL SHIFT]	^{D}
30	1E		^{E}
31	1F		^{F}
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	*
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	Ç̈	
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	Δ / Cursor left one word
197	C5	SHIFT ALT Left	Љ / Pan left
198	C6	SHIFT CTRL Left	Φ / Delete left one word
199	C7	SHIFT CTRL ALT Left	Γ
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	Ω
208	D0	Up	∏ / Cursor up
209	D1	ALT Up	Ψ / Scroll up
210	D2	CTRL Up	® / Search backward
211	D3	ALT CTRL Up	Σ
212	D4	SHIFT Up / Page Up	Θ / Top of screen
213	D5	SHIFT ALT Up / Home	ϒ
214	D6	SHIFT CTRL Up	†
215	D7	SHIFT CTRL ALT Up	‡
216	D8	Down	Ξ / Cursor down
217	D9	ALT Down	± / Scroll down
218	DA	CTRL Down	ç / 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	ӽ
228	E4	SHIFT CAPS LOCK	∂
229	E5	SHIFT ALT CAPS LOCK	∈
230	E6	SHIFT CTRL CAPS LOCK	$^{F}{}_{R}$
231	E7	SHIFT CTRL ALT CAPS LOCK	ɣ
232	E8	F1	ħ
233	E9	CTRL F1	⎩
234	EA	SHIFT F1 / F6	¦
235	EB	CTRL SHIFT F1	K
236	EC	F2	¼
237	ED	CTRL F2	½
238	EE	SHIFT F2 /F7	¾
239	EF	CTRL SHIFT F2	ɯ

Decimal	Hex	Keying	Display/Function
240	F0	F3	Ψ
241	F1	CTRL F3	⇒
242	F2	SHIFT F3 / F8	ρ
243	F3	CTRL SHIFT F3	σ
244	F4	F4	τ
245	F5	CTRL F4	υ
246	F6	SHIFT F4 / F9	√
247	F7	CTRL SHIFT F4	I√
248	F8	F5	ξ
249	F9	CTRL F5	…
250	FA	SHIFT F5 / F10	ξ̌
251	FB	CTRL SHIFT F5	ς
252	FC	SHIFT space / Insert	I / "Special" space
253	FD	SHIFT TAB	ʖ / 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 Minerva 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 Minerva and cannot be detected without special software.

clock

Minerva contains a real time clock, which runs when the Q68 is started. It obtains the current date and time from it's inbuilt battery backed real time 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 the day of the week as a string

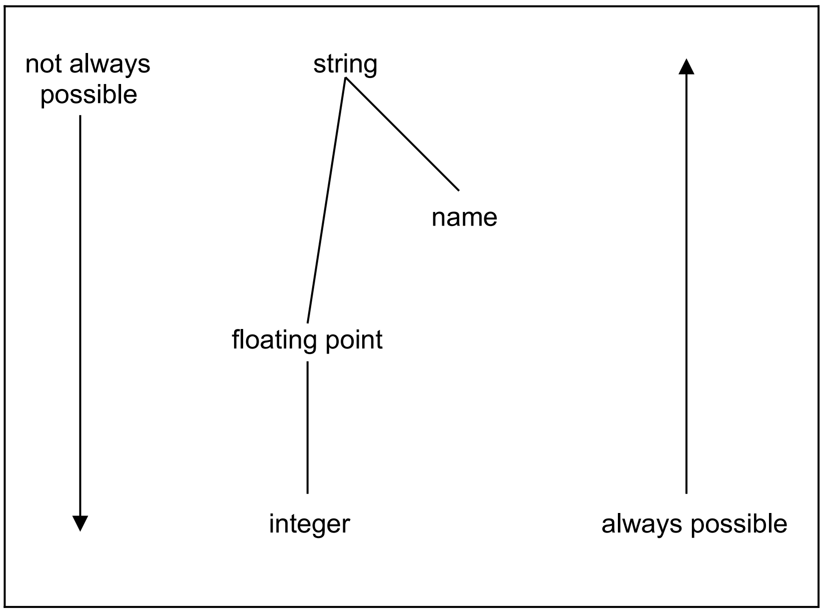
coercion

If necessary SuperBASIC 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 SuperBASIC 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 SuperBASIC **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 Minerva, see figure below. String is the most general type since it can represent integer data (almost exactly). The figure below shows the ordering diagrammatically. 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 SuperBASIC which would generate errors in most other computer languages. In general, it is possible to mix data types in a very flexible manner:

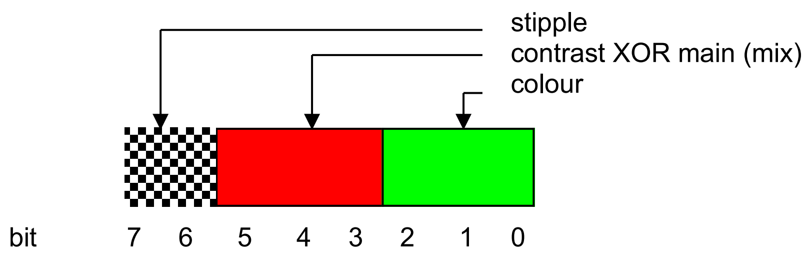
i. **PRINT "1" + 2 + "3"**
ii. **LET a$ = 1 + 2 + a$ + "4"**

colour

Colours in Minerva can be either a **solid colour** or a **stipple** - a mixture of two colours to some predefined pattern. Colour specification in Minerva can be up to three items: a colour, a contrast colour and a stipple pattern.

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*

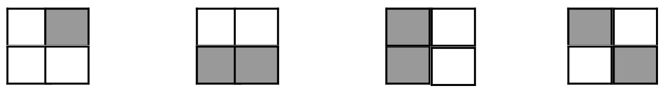
Background and *contrast* colours and *stipple* are each defined separately.

colours The codes for standard palette colours:

		bit		colour	
code	colour	pattern	composition	8 colour	4 colour
0	Black	0 0 0		black	black
1	Blue	0 0 1	blue	blue	black
2	Red	0 1 0	red	red	red
3	Magenta	0 1 1	red + blue	magenta	red
4	Green	1 0 0	green	green	green
5	Cyan	1 0 1	green + blue	cyan	green
6	Yellow	1 1 0	green + red	yellow	white
7	White	1 1 1	green + red + blue	white	white

Colour Composition and Codes

stipples Stipples mix a background and a contrast colour in a fine stipple pattern. Stipples can be used in Minerva 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.

communications
serial RS-232-C

The Q68 is fitted with a single PC standard male 9 pin SubD connector serial port called SER1, 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 serial port on the Q68 can operate at a 7 or 8 bit rate. There is no parity, nor hardware handshake.

9 pin	Name	Function	Direction
2	RXD	Receive Data	In
3	TXD	Transmit Data	Out
4	DTR	Data Terminal Ready Internally pulled up for serial mice	Out
5	GND	Signal Ground	-
7	RTS	Ready to Send Internally pulled up for serial mice	Out

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 the Q68 and the connected equipment are the same. The serial port on the Q68 can be set to operate at:

```
  1200
  2400
  4800
  9600
 19200
 38400
 57600
115200
230400   baud
```

The Q68's baud rate for the serial port is set by the **BAUD** command.

Any parity instructions set when opening a serial channel will be ignored.

Flow control determines how the Q68 and the peripheral device know when to communicate with each other. Flow control can be either:

Hardware	Not supported in the Q68.
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 the Q68 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 Q68 character set, and the printers characters set. See the **TRA** command.

Serial communications on the Q68 are 'full duplex', that is both receive and transmit can operate concurrently.

The parity and handshaking are selected when the serial channel is opened. (Not supported in the Q68)

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

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 SuperBASIC, 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 SuperBASIC. 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 SuperBASIC 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**

devices

A device is a piece of equipment on the Q68 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 disk file and stored. The file can then be printed at a later date. I/O on the Q68 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 the Q68. 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 SuperBASIC type name though it is also possible to build up the file name (device name) as a SuperBASIC string expression.

In summary the general form of a file name is:

identifier [*information*]

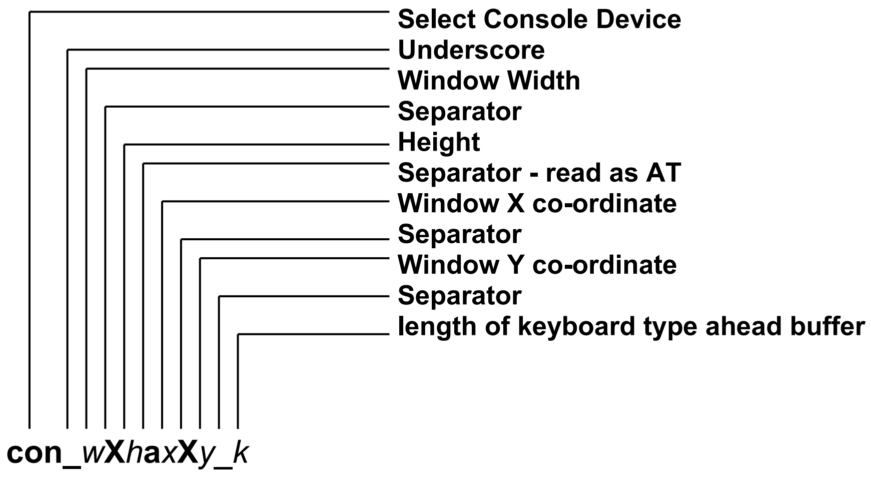
where the complete file name (including the extra information) conforms to the rules for a SuperBASIC identifier.

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	[*f*] handshaking	[*t*] translate
e – 7 bit + even	**i** - ignore flow control	**d** - direct output
o – 7 bit + odd	**h** – handshake CTS/DTR	**t** - translate
m – 7 bit + mark (1)	**x** - XON/XOFF	
s – 7 bit + space (0)		

[*c*] carriage return	[*e*] end of file
r - raw data	**f** - <FF> at end of file
c - <CR> is end of line	**z** – CTRL Z at end of file
a - <CR><LF> is end of line	
<CR><FF> is end of page	

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	[*f*] handshaking	[*t*] translate
e – 7 bit + even	**i** - ignore flow control	**d** - direct output
o – 7 bit + odd	**h** – handshake CTS/DTR	**t** - translate
m – 7 bit + mark (1)	**x** - XON/XOFF	
s – 7 bit + space (0)		

[*c*] carriage return	[*e*] end of file
r - raw data	**f** - <FF> at end of file
c - <CR> is end of line	**z** – CTRL Z at end of file
a - <CR><LF> is end of line	
<CR><FF> is end of page	

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	[*f*] handshaking	[*t*] translate
e – 7 bit + even	**i** - ignore flow control	**d** - direct output
o – 7 bit + odd	**h** – handshake CTS/DTR	**t** - translate
m – 7 bit + mark (1)	**x** - XON/XOFF	
s – 7 bit + space (0)		

[*c*] carriage return	[*e*] end of file
r - raw data	**f** - <FF> at end of file
c - <CR> is end of line	**z** – CTRL Z at end of file
a - <CR><LF> is end of line <CR><FF> is end of page	

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

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

PIPE[*IDin*] [X|P|T] *IDout* [_[l*ength*]] [K] Two ended Pipe device (first in, first out)

IDin indicates the input channel ID.
X is a separator when IDin, P, or T are not specified.
P indicates a permanent pipe.
T indicates a temporary pipe.
IDout indicates the output channel ID.
_length indicates an output pipe length in bytes. 0 indicates an input pipe.
K indicates that *length* in in kilobytes.

There are two additional pipe-like devices supported:

pipep	will start a copy of MultiBASIC.
pipet	is a device that always gives an EOF on read and discards output.

default: no default

example: **EXEC pipep**
OPEN #4, pipe_2048
OPEN #5, pipe_0

comment: See the PIPE section for more information on using pipes

QUB*n_name* Qubide container file drive File Access

n	QUB drive number
name	QUB drive file name
default:	no default
example:	**OPEN #9, qub1_data_file** **OPEN #9, qub1_test_program** **COPY qub1_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

SuperBASIC 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 SuperBASIC 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 SuperBASIC program and stored. These statements constitute a SuperBASIC program and will only be executed when the program is started with the **RUN** or **GOTO** command.

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

directory devices

Directory devices handle individual files, organised in directories (with at least one root directory). The drives QUB and WIN are used to access the hard disk container files on FAT32 partitions on the SDHC cards. More details can be found in the hardware-dependent sections of this manual.

display modes

The Q68 can operate in any of 8 different display modes. Only 3 of these are supported in Minerva4Q68.

The display modes are selected with the **DISP_MODE** command, and are numbered as follows:

Mode	Description
0	**QL 8 colour mode** This is the standard 256 x 256 pixels mode in 8 colours. In this mode you can also set mode 4, with the usual **MODE 4** keyword. This is then equivalent to setting **DISP_MODE 1**.
1	**QL 4 colour mode** This is the standard 512 x 256 pixels in 4 colours mode. In this mode you can also set mode 8, with the usual **MODE 8** keyword. This is then equivalent to setting **DISP_MODE 0**.
4	**Large QL 4 colour mode** This is a display of 1024 x 768 pixels in QL 4 colours mode (there is no mode 8 in this display mode).

Command	Function
DISP_MODE	sets the display mode
DISP_TYPE	returns the Minerva4Q68 display type

high resolution mode

From Minerva4Q68 v1.4 onwards, the Q68's 1024 x 768 4 colour mode is supported. Note that this mode has not been tested extensively so please use it with caution.

The 1024 x 768 4 colour mode is implemented using the **DISP_MODE** command with a subset of the SMSQ/E version. Currently, modes 0 (256 x 256 x 8), 1 (512 x 256 x 4), and 4 (1024 x 768 x 4) are supported.

Implementing the full range, including 65536-colour modes, would require a total rewrite of the screen drivers, including implementation of the GD2 colour schemes, which is far beyond the scope of the Minerva4Q68 project and are already available within SMSQ/E.

Note that Minerva's dual screen feature is not supported in 1024 x 768 mode, and trying to switch to **DISP_MODE 4** with dual screen enabled will produce a 'not complete' error. Please reboot first with dual screen disabled.

If you use the Pointer Interface in1024 x 768 mode, then some caution is required. The ptr_gen program needs to be patched to support the extended screen size and different screen buffer address. Thus, you must load it with some code like this (Toolkit II extensions required):

```
200 DEFine PROCedure patch_ptr
210 LOCal a,p,s
220   a=RESPR(FLEN(\ptr_gen))
230   LBYTES ptr_gen,a
240   s=0: PRINT "Patching pointer interface...";
250   FOR p=a TO a+FLEN(\ptr_gen) STEP 2
260     IF PEEK_L(p)=32768 AND PEEK_W(p+4)=128 AND PEEK_W(p+6)=512
        AND PEEK_W(p+8)=256 THEN
270       POKE_L p-4,HEX('fe800000'): POKE_L p,HEX('30000'): REMark
          buffer address and size
280       POKE_W p+4,256: POKE_W p+6,1024: POKE_W p+8,768: REMark
          line length, X size, Y size
290       s=1: EXIT p
300     END IF
310   END FOR p
320   IF s=1 THEN
330     PRINT "Success!": CALL a: LRESPR wman: LRESPR hot_rext
340   ELSE
350     PRINT "Failed!"
360   END IF
370 END DEFine patch_ptr
```

Note that you must switch to 1024 x 768 mode *BEFORE* activating the Pointer Interface. After this, you cannot switch back to the lower-resolution modes.

The functions **SCR_BASE**, **SCR_LLEN**, **SCR_XLIM** and **SCR_YLIM** return the base address, pixel line length in bytes, and X and Y limits of the current screen mode, like their SMSQ/E counterparts. In the current version, any parameters are ignored.

Command	Function
SCR_BASE	returns the screen base address
SCR_LLEN	returns the pixel line length in bytes
SCR_XLIM	returns the X limits of the current screen mode
SCR_YLIM	returns the Y limits of the current screen mode

dual screens

Minerva can operate in dual screen mode, if F3 of F4 are pressed at startup. The **MODE** command allows you to use both of these screens.

See the **MODE** command description in the keyword document for details on using the dual screens in Minerva.

error handling

Errors are reported by SuperBASIC 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) Not complete
An operation has been prematurely terminated (or break has been pressed).

(2) Invalid job
An error return from Minerva relating to system calls controlling multitasking or I/O.

(3) Out of memory
Minerva and/or SuperBASIC has insufficient free memory.

(4) 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) Channel not open
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.
SuperBASIC 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) 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) Drive full
A device has been filled (usually Floppy disk).

(12) Bad name
The file system has recognised the name but there is a syntax or parameter value error.

In SuperBASIC it means a name has been used out of context. For example, a variable has been used as a procedure.

(13) Xmit error
RS-232-C parity error.

(14) Format failed
Attempted format operation has failed, the medium is possibly faulty.

(15) Bad parameter
There is an error in the parameter list of a system or SuperBASIC procedure or function call.

An attempt was made to read data from a write only device.

(16) Bad or changed medium
The medium is possibly faulty

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

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

(19) Not Implemented

(20) Read only
There has been an attempt to write data to a shared, or write protected file.

(21) Bad line
A SuperBASIC 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 SuperBASIC to reset its internal state to the outer program level and so losing any procedure environment which may have been in effect.

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.

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.

The **WHEN** keyword is used to implement a sort of *implied subroutine* system, where the programmer doesn’t explicitly write a procedure call or **GO SUB** but lets it happen when the conditions are right, as it were.

Having said that, **WHEN ERRor** routines are executed when conditions are wrong, i.e. an un-trapped error has occurred. The syntax is:

```
WHEN ERRor : <statements>
```

or

```
WHEN ERRor
<statement>
<statement>
<statement>
<statement>
END WHEN
```

If an error occurs then the statements on the **WHEN ERRor** line, or between the **WHEN ERRor** and **END WHEN** lines, will be executed. Normal execution will then resume at the statement after the one that caused the error. SuperToolkit II users can use the improved **CONTINUE** and **RETRY** statements to resume elsewhere.

If an error occurs within the **WHEN ERRor** routines, then the program will halt with the usual error message, but with the additional information *during **WHEN** processing* added. The **WHEN ERRor** routine is added when it is encountered: thus errors in statements executed before this will cause errors as usual, and the recovery routine can be changed by passing through another **WHEN ERRor** block. The last such block encountered remains in force even after the program stops, so errors in the command line will cause a recovery attempt - you can turn this off by typing **WHEN ERRor** at the command line.

```
100 WHEN ERROR : PRINT “Whoops!”
110 PRINT “The answer isnt”,1/0
120 WHEN ERROR
130 PRINT “Eeek!”
140 END WHEN
150 PRINT 110
```

will thus print

```
Whoops!
Eeek!
```

and all subsequent error messages will be Eeek! until you type **WHEN ERRor** at the command line!

WHEN Variable

WHEN variable will execute a routine when a simple variable is assigned to. It does not work with arrays, nor when a variable is **INPUT** or **READ** into. A number of **WHEN** conditions can be set up for a variable, and you can of course have multiple **WHEN** variables.

If **WHEN** conditions overlap there is no guarantee as to which will be chosen.(???)

```
100 WHEN i=5:PRINT “i is five”
110 WHEN i>8:PRINT “i is big”
120 FOR i=1 TO 10:PRINT i
```

will print

```
1
2
3
4
i is five
5
6
7
8
i is big
9
i is big
10
```

You can get very silly with this facility

```
100 WHEN a=1:PRINT “a is one
110 WHEN a=2:PRINT “a is now two”:b=5
120 WHEN b=5:a=1:PRINT “b is five”:a=2
130 a=2
```

will print:

```
a is now two
a is one
b is five
```

and end up with a=2. Note that because the **WHEN** block at line 110 was already active when the last statement of line 120 is executed, is doesn't get re-entered. If you alter line 130 to b=5, you'll get:

```
a is one
b is five
a is now two
```

Provided the condition starts with a simple variable, it can be as complex as you like:
WHEN a>5 AND a<10 is valid.

expressions

SuperBASIC 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**

file types
files
All I/O on the Q68 is to, or from a 'logical file'. Various file types exist.

data	SuperBASIC 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**.

functions and procedures

SuperBASIC 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 SuperBASIC 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 SuperBASIC 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 SuperBASIC variables. The actual parameters must be an array, an array slice or a SuperBASIC 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 SuperBASIC can be used recursively. That is a function or procedure can call itself either directly or indirectly.

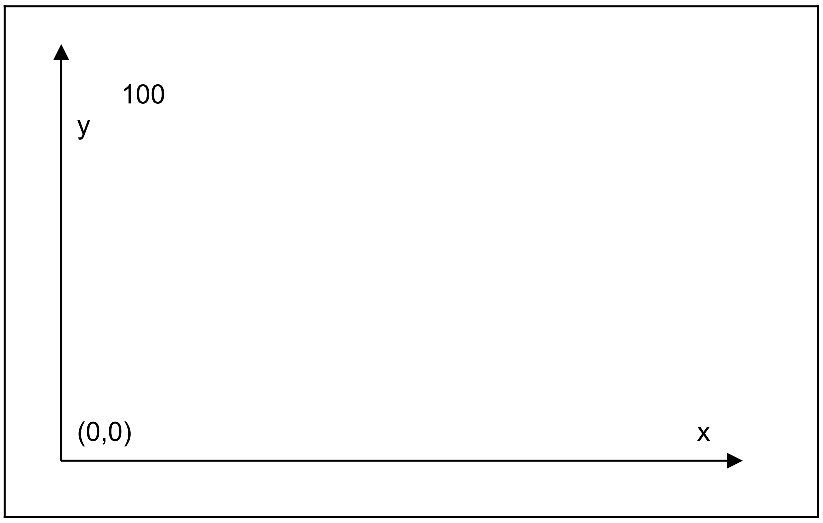
Command	Function
DEFine FuNction	define a function
DEFine PROCedure	define a procedure
RETurn	leave a function or procedure (return data from a function)
LOCal	define local data in a function or procedure

graphics

It is important to realise that in some display modes the Q68 screen has non-square pixels and that changing screen mode will change the shape of the pixels. Thus the graphics procedures will draw different shapes in different modes. Circles will be circular, in display modes 0 and 1, and elliptical in display mode 4.

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.

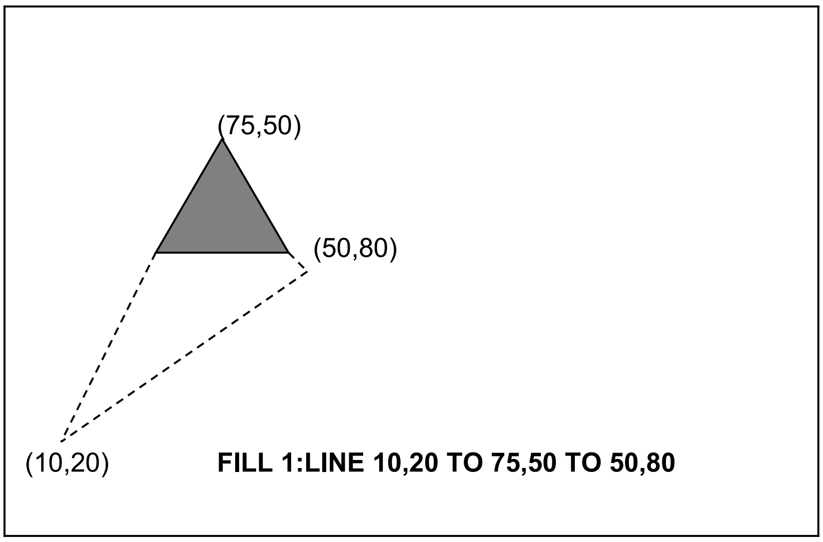
The graphics procedures are output to the window attached to the specified or default channel and the output is drawn in the **INK** colour for that channel.

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.

identifier

An SuperBASIC 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 SuperBASIC system to identify *variables*, *procedures*, *functions*, *repetition* loops, etc.

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

keyboard changes

A number of changes have been made to the keyboard, to improve usability and gain access to some of the new facilities.

The following list of keys did nothing (useful) in previous versions: while the functions they now perform are (where appropriate) retained on their original keys, these ones are hard-wired into the system and can't be modified by **POKE**s. This is especially useful for the new "next jobs" key, as **CTRL-C** is forever being zapped by unfriendly software, and never twice the same key either!

Keystroke	**Function**	**Old keystroke**
[CTRL] [ALT] [SPACE]	BREAK MultiBASICs	(none)
[CTRL] [TAB]	swap displayed screen	(none)
[CTRL] [ALT] [TAB]	screen freeze	CTRL-F5
[CTRL] [ALT] [SHIFT] [TAB]	Keyboard RESET	(none)
[CTRL] [ENTER]	compose character	(none)
[CTRL] [ALT] [ENTER]	keyboard queue	CTRL-C
[SHIFT] [CTRL] [ENTER]	CAPSLOCK	CAPSLOCK
[ALT] [CTRL] [SHIFT] [ENTER]	Call User routine	(none)

Compose characters
The only really non-obvious one of these is compose character, **CTRL-ENTER**. This allows you to type in that tricky foreign character you know is in there somewhere, but is it on **CTRL - =** or **CTRL-SHIFT-1** ? Now all you need do is type **CTRL-ENTER**, **A**, **:** for a-umlaut (an a with two dots), and so on. Where an upper-case version exists, shifting either of the two characters gives the upper-case result (or having caps lock on, of course). It's been tried to keep the combinations pretty obvious: \ and **/** combine with letters to give grave and acute accents, **:** for umlaut, and **^** (or 6) for circumflex. The quote key has been avoided, as it's not obvious whether it adds an accent (´) or umlaut(´´). Note that symbols (**^**, **:** etc.) are added correctly whether or not you press the **SHIFT** with them: you get an umlaut from **CTRL-ENTER**, **A**, **;** as well.

The compose table is currently as follows:

á	a/	ô	o^	ø	o!	¡	!!
à	a\	ú	u/	ü	u:	¿	??
â	a^	ù	u\	ç	c,	§	pp
ë	e:	û	u^	ñ	n~	¤	ox
è	e\	ß	ss	æ	ae	«	<<
ê	e^	¢	c!	œ	oe	»	>>
é	e/	¥	y-	α	aa	º	oo
ï	i:	`	\\	δ	dd	÷	:-
í	i/	ä	a:	θ	tt	←	the
ì	i\	ã	a~	λ	||	→	appropriate
î	i^	å	ao	µ	mm	↑	cursor
ó	o/	ö	o:	π	pi	↓	key
ò	o\	õ	o~	Φ	ph		

keyword

SuperBASIC keywords are identifiers which are defined in the SuperBASIC *Keyword Reference Guide*. Keywords have the same form as a SuperBASIC 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 SuperBASIC to recognise the keyword.

The set of SuperBASIC keywords may be extended by adding *procedures* to the Q68. It is a good idea to define these with their names in upper case and this will indicate their special function in the SuperBASIC system. Conversely, ordinary procedures should be defined with their names in lower case.

warning: Existing keywords cannot be used as ordinary identifiers within a SuperBASIC program.

maths functions

SuperBASIC 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 π
RND	generate a random number
RANDOMISE	reseed the random number generator

multiBASIC

There are very few differences between a MultiBASIC and the standard SuperBASIC interpreter, (job 0). A MultiBASIC can be started in exactly the same way as any other job, using **EXEC**, a *front-end* program, or one of the Qjump hotkey systems (not supplied with Minerva4Q68).

A new system vector allows an **EXEC**ed job to promote itself to being a SuperBASIC interpreter. This will inherit all the procedures and functions available to its parent interpreter: any others added to the parent subsequently will not be seen by the child interpreter, and any added to the child are only seen by it and its offspring, disappearing when it is removed.

To start a MultiBASIC, just enter the command **EXEC pipep**

You are now looking at another SuperBASIC interpreter which to all intents and purposes behaves just like the original with the noted exceptions below.

A MultiBASIC can have its job priority altered just like any other job and any toolkit extensions which relate to job control should work in the usual way.

Please note that you should load only SuperBASIC extensions into a MultiBASIC, unless you can guarantee that you'll never throw it away. Loading operating system extensions, such as QJump's Pointer Interface, is almost bound to cause problems if they disappear when the owner job goes away. Packages in the latter category include Lightning, and SuperToolkit II with the MDV extensions: SuperToolkit II without the MDV stuff, the Pointer Toolkit, and the Turbo Toolkit should be safe enough.

The MultiBASIC supplied has just one channel opened for it, which is used for both channels #0 and #1. If you want something that looks like an ordinary SuperBASIC interpreter, as seen at boot time, the following program will do the trick - note that it needs SuperToolkit II:

```
100 OPEN #0;con : OPEN #1;con : OPEN #2; con
110 WMON 4
```

Removing a MultiBASIC

A MultiBASIC will remove itself if it encounters an error while reading a new command from its primary command channel, #0. You can therefore get it to go away by typing CLOSE #0 at it.

Advanced use

For more advanced use, you can use QX or EX to pass channels and/or command string.
If the last character of the command string is the "ROM" marker (an exclamation mark) it is removed from the string and the interpreter will start up with only the original ROM names, instead of inherited names. The remaining command string is then scanned for the "file" marker (a greater-than sign), and if it's got it, the first part is opened as an input command channel, and the rest is shuffled down.

The command string, what's left of it, becomes the CMD$ variable in the interpreted basic.

Channels passed:

None:	If no file marker in the command string, a single window is opened for both #0 and #1
One:	Slotted in as both #0 and #1
Two:	Become #0 and #1
More:	First two become #0 and #1, #2 is missed out, and the rest go in as channels #3 onward.

E.g. A filter to replace strings in a file:

```
100 a$='':
110 i%='/' INSTR cmd$
120 IF i%THEN
130  a$=cmd$(i%+1TO)
140  cmd$=cmd$ (TO i%-1)
150 END IF
160 c%=LEN(cmd$)
170 REPeat lp1
180  IF EOF(#0) THEN EXIT lp1
190  INPUT#0,i$
200  IF c% THEN
210   REPeat lp2
220    i%=cmd$ INSTR i$
230    IF NOT i% THEN EXIT lp2
240    PRINT i$(TO i%-1);a$;
250    i$ = i$(i% + c% TO)
260   END REPeat lp2
270  END IF
280  PRINT i$
290 END REPeat lp1
999 IF VER$(-1) THEN POKE\48\0,-1
```

Save this in a file called *flp1_c_ba*s, then use:

EX flp1_multi,flp1_in,flp1_out;'flp1_c_bas > fred/jim'

to convert all occurrences of *fred* in *flp1_in* to *jim*, writing the result to *flp1_out*.

Resident MultiBASIC
An additional MultiBASIC file is provided on the Minerva Utilities Disk (not supplied with Minerva4Q68) MultiB_REXT which when loaded with **LRESPR** or the following:

base=RESPR(344) :LBYTES flp1_MultiB_rext,base :CALL base

will add the SuperBASIC extension **MB** which will invoke a new copy of the MultiBASIC job. This is allows you to have MultiBASICs available without needing to **EXEC** them from a disk, but it is not quite as convenient as having them resident on a hotkey.

The Minerva Utilities Disk may be found at https://dilwyn.qlforum.co.uk/qlrom/min198utils.zip

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 SuperBASIC 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 SuperBASIC 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).

The full syntax for the pipe device is now:

PIPE [*IDin*] [X|P|T] *IDout* [_[l*ength*]] [K]

IDin, *IDout* and *length* are all decimal values in the range -32768 to 32767 and if omitted, they default to zero, with one exception. If one of "X", "P" or "T" is given, and *IDin* is present, and not zero, *IDout* will default to (or a negative value be forced to) the same value as *IDin*.

If "K" is given, the "length" is multiplied by 1024 to give the actual length of pipe to be created. (Note: the length of a pipe is the exact number of characters that it can have written into it before it will be full, and need someone to start reading them out.)

Old style pipes
If none of "X", "P" or "T" is given, and *IDin* is omitted, or zero, the old style pipes are used.

Firstly a channel must be opened to write to the pipe, specifying a length greater than zero for the length of the queue to be created. A second channel must then be opened to read from the pipe, by giving a length less than or equal to zero, or omitted. The QDOS channel number of the first channel must be passed in D3.w when the second channel is opened.

An aside: on a totally bare machine, it was just possible to connect up pipes, provided you didn't mind losing one of #0, #1 or #2. e.g.

OPEN#2;'pipe_100' : OPEN_NEW#6;'pipe'

would work, as the open code IO.NEW is 2, and QDOS channel no 2 just happened to be associated to #2.

The facility to specify long pipes via using "K" is available even on the old style pipes. Also, the connection is much more thoroughly checked. Only one input channel at a time is permitted and connecting input pipes to one another is rejected.

New style pipes
These come in when either *IDin* or *IDout* is non-zero.

The first channel opened to an "ID" must specify a length greater than zero for the length of the queue (or queues) to be created. On any subsequent opens using this particular "ID", the length is totally ignored.

There is no limit to the number of channels inputting from and/or outputting to the same ID pipe.

If *IDout* is zero (or omitted), and one of "X", "P" or "T" is given, the channel will be read-only.

If *IDin* is omitted, the channel will be write-only.

If both *IDin* and *IDout* are non-zero (or *IDout* was made to duplicate *IDin*, as described above), the channel will provide data from *IDin* and data sent to it will go to *IDout*. The can be the same "ID", in which case a single channel can be used as a "first in, first out" circular queue.

Normally, when the last channel capable of writing to an "ID" has been closed, the queue is marked as at "end-of-file" and becomes anonymous. The "ID" is then available for re-use. When there are no channels at all connected to an anonymous queue, any data still in it is lost and its memory is returned to the system.

The "X" is just to separate *IDin* and *IDout*, but a "P" (permanent) will ensure that the pipe (or both pipes) are preserved even if all channel connections are closed. A "T" (temporary) will revert from this state.

Should one wish to open a channel to two distinct "ID"s, and want their lengths to be different, one has to has to open a dummy channel to one of "ID"'s first, in order to define its length. The required channel can then be opened to create the queue for the second "ID", and the dummy channel is then closed. Alternatively, only a single channel need be used, if the "P" and "T" flags are used. e.g.

OPEN#3;'pipe1p_100' : OPEN#3;'pipe1t2_20'

As a recommendation, a job which is creating pipes should incorporate its QDOS job number in the "ID"s that it uses.

The suggested system is to use the hundred "ID"s starting at the jobs own job number times a hundred. e.g.

'pipe'&j&'12'	where "j" is calculated by **j=VER$(-1) : j=j-INT(j/65536)*65536"** **j=(PEEK_L(!!100) - PEEK_L(!!104))DIV 4)** or any other convenient way.

Throwaway pipe
"pipet" is a device which will permanently give "end-of-file" on reading and will discard any output to it. Any number of channels may be open to this simultaneously.

Program pipe
"pipep" is device that is read-only, and contains a copy of MultiBasic, suitable for **EXEC[_W]** (or anything else that likes a serial stream input with a proper header showing the type executable, etc.) Any number of channels may be open to this simultaneously and they will each receive the full data.

Some examples:

OPEN#3;'pipe2x1_100' : OPEN#4;'pipe1x2' : EXEC'pipep',#4

You can drive your own "floating" copy of SuperBasic by PRINTing commands to #3 and seeing what comes back by INPUTing from the same #3.

A couple more examples:

PIPE1X_10	an input only pipe
PIPEX1_10	an output only pipe

A limited semaphore:

OPEN#3;'pipep1_10' : PRINT#3;'xy' : CLOSE#3

This has an initial value of three and a limit of 10. It can be useful for resource management, etc. Any program can come along and obtain a unit with:

OPEN#3;'pipe1' : a$=INKEY$(#3) : CLOSE#3

To release a unit:

OPEN#3;'pipe1' : PRINT#3 : CLOSE#3

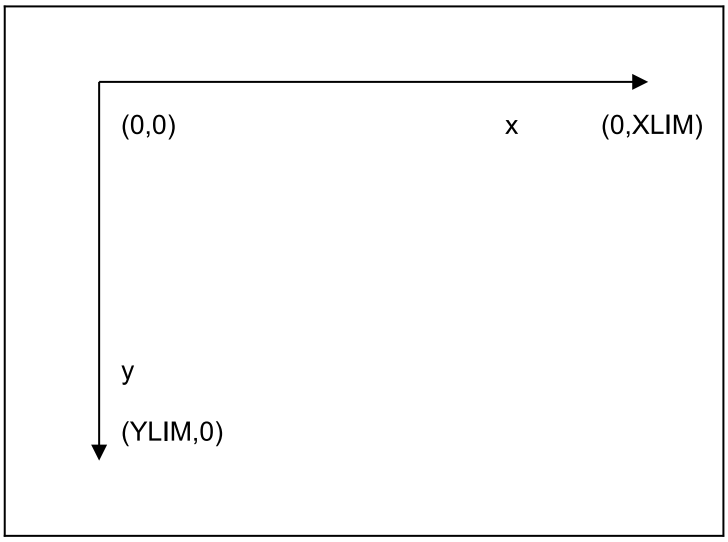
To finally discard the semaphore:

OPEN#3;'pipe1t ': CLOSE#3

pixel coordinate system

The **pixel coordinate system** is used to define the positions and sizes of windows, blocks and cursor positions on the Q68 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 SuperBASIC program consists of a sequence of SuperBASIC 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
Minerva

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

The Sinclair QL used a version of BASIC known as SuperBASIC. The Minerva and it's SuperBASIC are descendants of the QL SuperBASIC, and QDOS.

Minerva4Q68 includes all the QL SuperBASIC commands, and the commands which have been provided to support the various add-on drivers. Minerva supports 99.9% of SuperBASIC.

Minerva is a QDOS like operating system that was developed as a replacement ROM for the original QL computer. It included many improvements and bug fixes of the original Sinclair ROM's.

qub
directory device

This device allows you to the read the first (!) partition of a container image file formatted the Qubide way. Hence, each drive corresponds to one container file on the card, This is just like the WIN drive. The purpose is mainly for you to be able to get data off the Qubide drive and onto a proper WIN drive. You should not operate a Qubide type drive as your main storage system, use the WIN drives for that..

So basically, this device behaves just like the WIN device, except that it uses different container files.

The device is called QUB and there are 8 drives. Like the WIN device, you must indicate for each drive the name of the container image file and the card it is on. Again, sensible names have been preconfigured.

Assigning the QUB drives is done by using the Menu Config program on the Q68_ROM.SYS or the Min4Q68_rext file. QUB drives cannot be assigned from within Minerva4Q68.

repetition

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

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

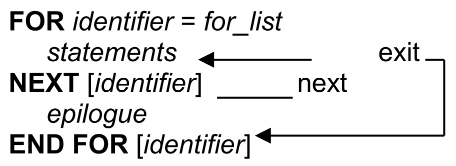
These two constructs are used in conjunction with two other SuperBASIC statements:

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**.

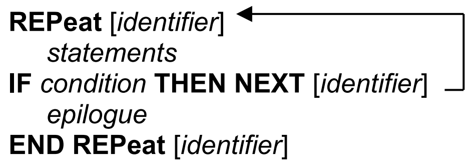
Processing an **EXIT** will pass control to the statement after the **END FOR** or **END REPeat** selected by the **EXIT** statement. **EXIT** can be used to exit through many levels of nested repeat structures. **EXIT** should always be used in **REPeat** loops to terminate the loop on some condition.

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 SuperBASIC statements which are executed on some special condition arising within the loop:


```
FOR identifier = for_list
    statements
NEXT [identifier]
    epilogue
END FOR [identifier]
```

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:


```
REPeat [identifier]
    statements
IF condition THEN NEXT [identifier]
    epilogue
END REPeat [identifier]
```

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.

SDHC cards

The Q68 is supplied with two SD card slots. These slots should only be used with SDHC type memory cards for mass storage in the Q68. Simple SD cards are not compatible.

The left card socket is socket 1, the one on the right is socket 2. Cards inserted into the left socket are referred to as **card1**, the card inserted into the right socket will be referred as **card2.**

For the cards to be useful, they must be partitioned, and the **first** primary partition must be formatted in the FAT32 format (this cannot be done on the Q68). The different files the Q68 needs must be put into that partition. This should be possible from any machine running an OS that can read/write SDHC cards (Linux, Windows, macOs): just copy the files to the card.

The Q68 always tries to start up from card1, by loading the operating system from that card. Once Minerva is loaded, it will follow its own usual boot process, normally running the boot file found on win1_.

container files

The Q68 uses container files in the FAT32 partitions of SDHC cards to represent QDOS style file systems. It supports the qxl.win and Qubide type container files.

The qxl.win type container files as used as the main mass storage devices in the Q68. These files must lie in the first primary partition on the SDHC card, which must be formatted with a FAT32 file system. Moreover, these container files must be located within the first 16 directory entries of this FAT32 formatted partition. This is also true for the file containing Minerva itself.

Special precautions must be taken when writing the container and OS file(s) themselves to the card. Indeed, Minerva expects a container file not to be fragmented in the FAT32 file structure. It assumes that, once it has found the beginning of a container file, the rest of that container file lies in contiguous sectors on the card. This is also true for the Minerva binary file (named "Q68_ROM.SYS") itself. Thus the files on the cards must not be fragmented.

The best way to achieve this is to make sure that, before writing the Minerva binary file and the container files, the card is freshly formatted. Then write each container (or other) file, one after the other, immediately after formatting the card.

Hence, you should dedicate a card solely for the purpose of using it with the Q68.

> **Do not** drag and drop several files onto the card at once.
> **Do not** delete files from the card – always format it.

It is **very much** recommended that you read the section **avoiding fragmentation** to make sure you treat the card as you should.

naming scheme

The file name of a container or OS file MUST be in "8.3" format, i.e. a name of 1 to 8 characters, possibly followed by a decimal point and a three letter extension. Missing letters are filled up with spaces. The name and extension must be in upper case and the extension, if present, must be separated from the name by a period (".").

Please only use plain ASCII characters for the name and no accented characters, i.e. the letters A-Z and numbers 0-9.

In all commands or configuration items where you must give or configure a name, Minerva tries to help you as much as possible. Names are automatically converted into upper case and correctly formatted, so that “qlwa.win” would automatically be converted to “QLWA .WIN”. However, a “_” is not converted to a “.”.

initialising a card

With the Q68, before you can use drives on an SDHC card, the card must be initialised (it would actually be more accurate to say that the card reader that a card is in must be initialised, i.e. put in a state where it will read a card). Card1 is automatically initialised at boot time.

By design, card2 is not initialised at boot time, though this will depend on you configuration options. If it is not initialised, you have to initialise it yourself. You can do this with the supplied **CARD_INIT** command. The card itself is not touched by this command (it is not formatted, written to or anything).

Command	Function
CARD_INIT	initialise the card reader

swapping cards

As a general rule, cards may be swapped in and out, even when the system is running, but this is not a recommended practice. However, if you insist on doing this, you must be aware of a few rules:

1 – Do not remove a card when there are files still open to a drive and certainly not whilst the machine is reading/writing to a card. If you remove a card whilst there are still files open or files being written, data loss WILL (not “may”) occur. Note that you will NOT be able to write the missing data to the card even if you reinsert it immediately after having removed it from the socket.

2 – When a card is removed from its socket, the card reader in that socket becomes uninitialised. Before using the new card that you just inserted, you must initialise it, with **CARD_INIT** as described above. This is true whether you insert a new card or re-insert the old one that was just removed.

avoiding fragmentation

Minerva for the Q68 expects that all container files (i.e. files for WIN drives, and also for QUB drives) lie in contiguous sectors on the SDHC card. If this is not the case, the file is said to be “fragmented”. Fragmented files are deadly on the Q68 under Minerva: Minerva assumes that, once it has found the beginning of a container file, the rest of that container file lies in contiguous sectors on the card, and it will cheerfully write into those contiguous sectors which it deems still belong to that file. If the file is fragmented, these contiguous sectors may not belong to it but to another file, **which will thus be irretrievably corrupted**.

This is also true for the Minerva binary file (named "Minerva.bin") itself.

So, special precautions must be taken when writing the container and OS file(s) themselves to the card. The best and recommended way to achieve this is to make sure that, before writing the Minerva binary file and the container files, the card is freshly formatted. Then, one after the other, write each container file to the card immediately after formatting the card.

Hence, you should dedicate a card solely for the purpose of using it with the Q68.

Note: practise has shown that in most cases it may not be necessary to reformat the card. You could also delete every single file on the card before copying new files onto it. Under no circumstances, however, should you only delete files selectively: this may leave “holes” in the file allocation table and this lead to fragmented files (see below). However, the recommendation still is to format the card and not just to delete all files from it.

When copying several files to the freshly formatted card, make sure that the copy process of each file is finished before you start that for the next file. If not, it may happen that the two copy processes write concurrently to the card, which could mean that the sectors for the two files interleave. Depending on the operating system you use (linux, windows, mac os) if you drag several files to the card at once, several concurrent copy processes might be started which might lead to file fragmentation. So, to avoid this, just drag the files to copy one after the other.

Moreover, never just delete a single file -be it a container file or any other file- from the card, but always format it (or at least delete ALL files from the card), and then write the files to the card again: If you delete a single file from the card and later write another, bigger, container file to the card, it is possible that part of this container file will lie in the sectors previously occupied by the deleted file, and the rest in previously unoccupied sectors. This file would then be fragmented and not lie in contiguous sectors on the card: a recipe for a disaster.

If a container file becomes fragmented, you WILL experience data loss, and other files on the card might also be irrecoverably damaged!

summing up

The first primary partition on a SD card must be formatted as FAT32.

Container files, and the operating system, must be in the above FAT 32 partition.

There should not be more than 16 container files (including the operating system container file) in the FAT 32 partition.

Files should be copied to the SD card one at a time.

Never delete container files, then copy more container files to the FAT32 partition of the SD card. Copy the container files you wish to keep off of the SD card, Format it, then copy the required container file back onto it.

OS and container filenames

There are several types of files that lie on an SDHC card during normal use of the Q68 with Minerva. All of these must adhere to the 8.3 naming scheme as set out above.

The Minerva file itself, a “naked” file, containing just Minerva itself.
In this case, the file **MUST** necessarily be called “**Q68_ROM.SYS**”.

The WIN device container files.

These should be called **QLWA**x**.WIN** where x can be any number between 0 and 9999, or be omitted.

It is recommended, but not mandatory, that you stay with this naming scheme. Since the names of the container files are configurable, you may basically call them whatever you like, provided you adhere to the 8.3 naming rules.

The QUB device container files.

These should be called **QL_BDI**x**.BIN** where x can be any number between 0 and 99. Again, it is recommended but not mandatory that you stay with this naming scheme.

However, since the names of the container files are configurable, you may basically call them whatever you like, provided you adhere to the 8.3 naming rules.

There again, a sensible default naming scheme has been devised:

QL_BDI.BIN on card1 for qub1_
QL_BDI.BIN on card2 for qub2_
QL_BDI3.BIN and QL_BDI4.BIN on card1 for qub3_ and qub4_.
QL_BDI5.BIN to QL_BDI8.BIN on card2 for qub5_ to qub8_.

Please remember that the qub device should not be your main storage system for the Q68.

serial port support

From Minerva 4Q68 version 1.6 onwards, the Q68's serial port is supported using a new driver in the ROM image. It offers the following features:

Configurable port name; default SER1 but can be changed using the **SER_USE** command

Alternative port names for transmit- and receive-only channels (STXx/SRXx), for use with SERnet

Baud rate configurable from 1200 to 230400 bits per second (using the normal BAUD command)

Flow control using XON/XOFF protocol with optional data transparency (between two Q68s or Q68 and QIMSI)

Configurable transmit- and receive buffers using **SER_BUFF** command

The Q68's serial port is much faster than the original QL's SER ports, but unfortunately lacks CTS/RTS lines so all flow control has to be done in software using XON/XOFF handshake.

The original QDOS/Minerva driver has only fixed-size buffers of 81 bytes, which is not adequate for handling high speeds. SMSQ/E, by contrast, has buffers of configurable size, and by default uses dynamic-size transmit buffers which can grow to insane size (probably designed to send files in quick succession to a printer). Unfortunately, all current versions of SMSQ/E do not support the XON/XOFF protocol even though the driver accepts 'X' as option on channel opens or as parameter to the **SER_FLOW** command, so sending or receiving files from or to the Q68 at full speeds will more or less lead to data corruption.
Reliable transfers are possible using SERnet (https://dilwyn.qlforum.co.uk/tk/sernet.zip; please use v2.25 as v3 will not work with Minerva). When using default buffer size, it is not necessary to enable XON/XOFF flow control, so specifying SRX1I/STX1I as device name will be sufficient.

Using SERnet, You are able to achieve throughputs up to 8.5K bytes at 115200 bps, which is twice as fast as the original QLAN network.

Command	Function
SER_USE	substitute an existing SER port
SER_FLOW	sets flow control
SER_BUFF	sets buffer sizes
SER_ROOM	sets the receive buffer's threshold for asserting flow control.
SER_CLEAR	clears both input and output buffers

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

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

SuperBASIC	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.

Minerva

Minerva is an operating system based on QDOS, and supervises:

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

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

system calls
System calls are processed by Minerva in **supervisor mode**. When in supervisor mode, Minerva 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 Minerva system vectors with appropriate parameters in the processor registers. The action taken by Minerva 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

Minerva supports a multitasking environment and therefore a file can be accessed by more than one process at a time. The Minerva filing sub-system can handle files which have been opened as **exclusive** files or as **shared** files. A shared file cannot be written to. Q68 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 Minerva are the same as the device names required by SuperBASIC and are discussed in the concept section *devices*. The collection of standard devices supplied with the Q68 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. SER, WIN) and then accessed in the same way as any other Q68 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 Minerva 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

Minerva 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.

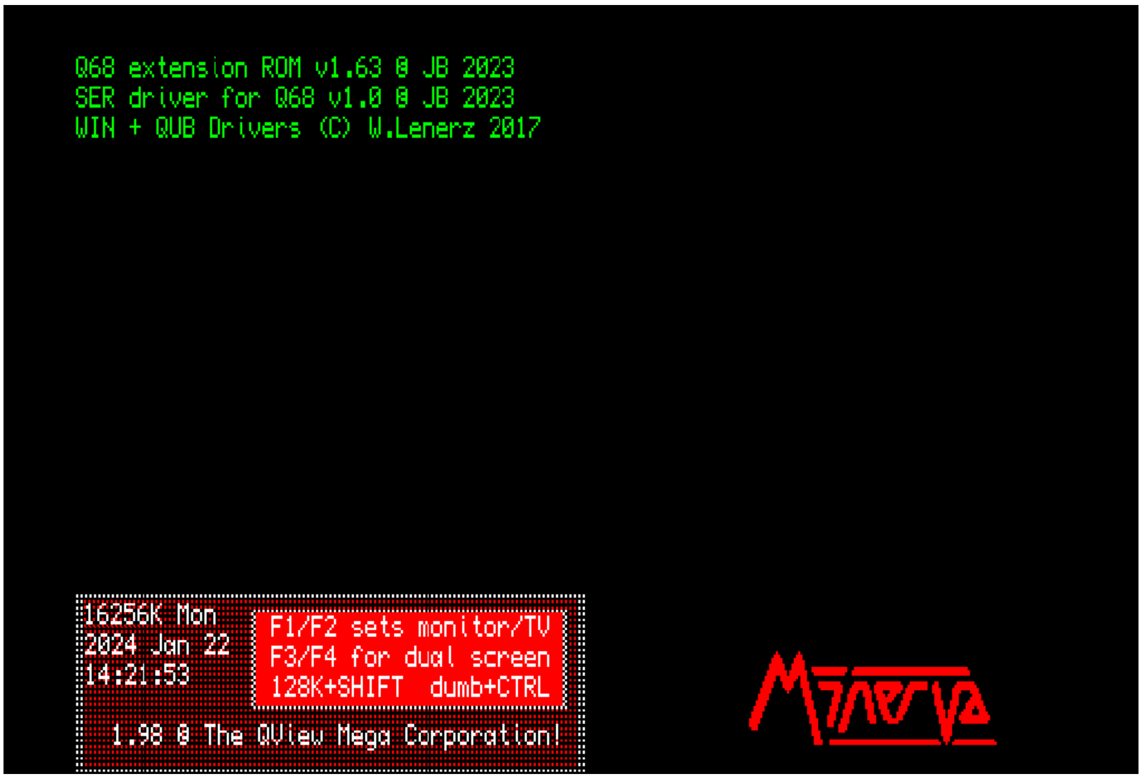
The x-origin and the y-origin (the last data item) should be 272 and 202 in monitor mode.

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

sound

Currently, sound is not supported by Minerva4Q68. It may be incorporated at a later date.

start up
Immediately after starting or resetting, Minerva will display the start up screen.



```
Q68 extension ROM v1.63 © JB 2023
SER driver for Q68 v1.0 © JB 2023
WIN + QUB Drivers (C) W.Lenerz 2017

16256K Mon      F1/F2 sets monitor/TV
2024 Jan 22     F3/F4 for dual screen
14:21:53        128K+SHIFT  dumb+CTRL

  1.98 © The QView Mega Corporation!
```

Pressing F1 will take you to Monitor mode (4 colours), and F2 will take you to TV mode (8 colours).

Pressing F3 or F4 will make Minerva restart as above, but with the second screen enabled. This also moves the system variables, which may cause problems for badly behaved software that assume where the system variables are located.

Pressing **SHIFT** and a function key will cut the amount of available memory to 128K, the same as an unexpanded QL.

Pressing **CTRL** and a function key omit the extension ROM scanning. **Avoid using this**, as you will loose the WIN, QUB and SER devices, and the keyboard.

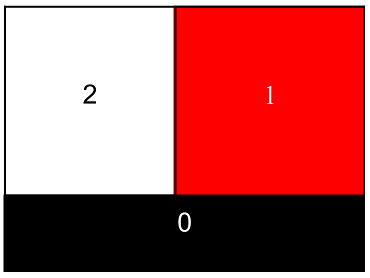
After pressing F3 or F4, pressing the screen switch key **CTRL-TAB** should now give you a blank screen instead of the pretty coloured dots and things that appear when you screen switch on a single screen Minerva.

auto-start
If you don't press F1 or F2 within fifteen seconds of the boot screen appearing, the system will start anyway, pretending that you've Just pressed the F1 key.

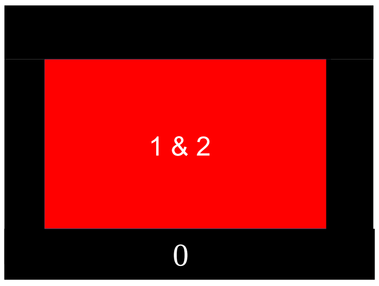
After pressing a function key, or auto-start, the monitor or TV screen is displayed.

The Minerva has the ability to 'boot' itself up from programs contained in win1_. If it contains a file called **BOOT** it is loaded and run.

default screen
The Q68 has three default channels which are linked to three default windows.



Monitor



Television

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.

statement

An SuperBASIC statement is an instruction to the Q68 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 SuperBASIC processes the statement immediately. If the statement is preceded by a line number then the statement becomes part of a SuperBASIC program and is added into the SuperBASIC program area for later execution.

Certain SuperBASIC 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 SuperBASIC 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 SuperBASIC. 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 SuperBASIC 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 SuperBASIC 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 SuperBASIC system.

Usage type 1 File and variable comparison
type 2 SuperBASIC <, <=, =, >= ,>, **INSTR** and <>
type 3 SuperBASIC == (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 SuperBASIC 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             a sequence of letters
               with a letter          digits and underscores
                                      i.e. repeat something
                                      which is optional
```

turtle graphics

SuperBASIC 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 the Q68 are large container files stored on an SDHC card. The files usually have the suffix “.WIN” but anything else is fine, too.

The corresponding Minerva device is called "WIN" and, potentially, you may have up to 8 different drives for this device, called "win1_" to "win8_".

The name and directory can be configured separately in the Menu Config program.

Each WIN drive can point to one container file lying indiscriminately on SDHC card one or two.

For each WIN drive, you must set the name of the container file, and the number of the card on which this file is to be found. You may do so by configuring this with the standard Menu Config program.

safety precaution
Do not point two different WIN drives to the same container file on the same card. For the time being, the system doesn't stop you from doing so, but data loss and file corruption WILL (not "can") occur as a result!

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