SMSQ/QXL

Introduction

SMSQ/QXL is based on the SMS kernel which was designed to provide a QDOS compatible interface. The kernel has been modified to improve compatibility with most of the "dirty tricks" which QL programmers were either forced to use or used to satisfy their perverted sense of fun.

The kernel itself (memory management, task management, scheduling, and IO) has also been extended to provide facilities which were not available with QDOS. It is now an over-inflated 10k bytes. Despite this inflation, the SMSQ operating system kernel remains more efficient than the old QDOS kernel.

SuperBASIC has been replaced by SBASIC which is a threaded code interpreter which executes at speeds more often associated with compiled SuperBASIC than interpreted SuperBASIC. There is no longer any need to compile SuperBASIC programs: you can just EXECute them.

In addition, SMSQ/QXL is supplied with entirely new filing system device drivers which allow "foreign" disk formats to be recognised and new formats to be added "at run time".

New and Modified Facilities.

SMSQ/QXL includes all the QL SuperBASIC commands, the TK2 commands (activated by the TK2_EXT command - this manual does not concern itself with the standard SuperBASIC or TK2 commands.) SMSQ/QXL supports 99.9% of SuperBASIC. SMSQ/QXL supports all the devices which were supported by the drivers supplied with the Trump and the GOLD card.

There are, however, a number of significant new facilities or improvements, some of which may be familiar to some users.

Facility	**Usage or Difference**	**Page**
$nnn %nnn	Hexadecimal and binary values accepted	6
ATAN	ATAN (x,y) yields four quadrant result	14
BAUD	Independent baudrates	22
BGET BPUT	Transfer multiple bytes to and from strings	14
CACHE_ON _OFF	Turn internal caches on or off	5
DEV	A defaulting filing system device	24
DEV_LIST	Lists the current DEVs	25
DEV_NEXT	Enquires the next DEV for a DEV	25
DEV_USE	Sets the real device for a DEV	24
DEV_USE$	Enquires the real device for a DEV	25
DEVTYPE	Find the type of device open as a channel	9
END FOR END REPeat	Do not need names	7
EX EW EXEC EXEC_W	Extended to execute SBASIC programs	17
EPROM_LOAD	Loads and initialises a "QL EPROM cartridge"	12
EXIT	Does not need a name	7
IF	Multiple nested inline IFs. Nesting is checked	6

IO_PRIORITY	Set the priority of IO retry scheduling	5
JOB_NAME	Sets the Job name for SBASIC jobs	17
KBD_TABLE	Uses international codes to set keyboard tables	21
LANG_USE	Sets the message language	21
LANGUAGE ($)	Language enquiry	21
LOAD LRUN	Accept QLOAD _SAV files and save filename	11
LBYTES	Accepts channel number in place of name	12
LRESPR	If used to load extensions within an SBASIC job other than job 0, the extensions are private to that job	18
MERGE MRUN	Accept QLiberator _SAV files	11
NEXT	Does not need a name	7
NUL	A bottomless bin for output or endless input	23
PEEK etc.	Extended to access system and SBASIC vars	13
PEEK$	PEEKs multiple bytes	12
PIPE	Named or unnamed pipes for inter task comms	23
POKE etc.	Extended to access system and SBASIC vars	13
POKE$	POKEs multiple bytes	12
PROT_DATE	Protect the real time clock	14
QLOAD QLRUN	Qliberator compatible quick load for _SAV file	11
QMERGE QMRUN	Qliberator compatible quick merge for _SAV file	11
QSAVE QSAVE_O	Qliberator compatible save to _SAV file	11
QUIT	Removes this SBASIC job	17
REPeat	Does not need a name	7
SAVE SAVE_O	Use previously defined filename, update version	11
SBASIC	Starts an SBASIC daughter	16
SBYTES SBYTES_O	Accepts channel number in place of filename	12
SELect	Both integer and floating point SELects	6
SEXEC SEXEC_O	Accepts channel number in place of name	12
SLUG	Slows the machine down	5
TRA	Language selectable and language independent	22
VER$	Minerva compatible	14
WMON WTV	Allow the SBASIC windows to be offset	17

SMSQ Performance

In general, SMSQ is more efficient than QDOS. There are, however, a number of policy differences which are either accidental because, unlike other "QDOS compatible" systems SMSQ is not based on QDOS but is completely re-designed, or deliberate because certain QDOS policies have shown to be less than ideal.

In particular, the IO retry scheduling policy is completely different. This results in a very much higher priority for retry operations which greatly improves the responsiveness of a heavily loaded system at the cost of a modest reduction in crude performance (typically 10%). If crude performance is important to you, you can reduce the the IO priority to QDOS levels.

IO_PRIORITY

The IO_PRIORITY (*priority*) command sets the priority of the IO retry operations. In effect, this sets a limit on the time spent by the scheduler retrying IO operations.

A priority of one sets the IO retry scheduling policy to the same as QDOS, thus giving a similar level of response but with a higher crude performance.

IO_PRIORITY 1	*QDOS levels of response, higher crude performance*
IO_PRIORITY 2	*QDOS levels of performance, better response under load*
IO_PRIORITY 10	*Much better response under load, degraded performance*
IO_PRIORITY 1000	*Maximim response, the performance depends on the number of jobs waiting for input.*

CACHE_ON CACHE_OFF

The performance of the more powerful machines depends on the use of the internal cache memory. For the MC680x0 series processors, the implementation of the caches is less than perfect. As well as introducing unnecessary overheads on operating system calls (slightly improved in the MC68040) the MC680x0 cache policy is incompatible with certain programming techniques. It may, therefore, be necessary to disable the internal caches.

No provision is made for disabling the external caches (where these exist) as none of these external caches seem to suffer from the design flaws of the MC680x0 series.

CACHE_OFF	turn the caches off to run naughty software
CACHE_ON	and turn back on again

SLUG

The designers of SMSQ have spent much time and effort trying to make the system fairly efficient. Their efforts seem not to be appreciated. Some people will always complain!

SLUG (*slug factor*) will slug your machine by a well defined factor.

SLUG 2	Half speed ahead
SLUG 5	Dead slow
SLUG 1	Full ahead both

SBASIC / SuperBASIC Language Differences

Some differences between SBASIC and SuperBASIC may be accidental. There are, however a number of known, deliberate, differences. Most of these differences are extensions to SuperBASIC. In some cases, however, limitations have been introduced to reduce the chances of difficult-to-track-down program errors.

Hexadecimal and Binary Values

Hexadecimal and binary values may be included directly in SBASIC source. Hexadecimal values are preceded by a $. Binary values by a %.

```
IF a% && %1001                      Check bits 3 and 0 of a%
IF PEEK_L ($28000) = $534D5351      Check if SMSQ (very naughty)
```

IF Clauses

Multiple "in-line" IF clauses can be nested on one line.

SBASIC checks for incorrectly nested IF clauses.

SELect Clauses

SELect clauses may SELect an action on the value of an integer variable (integer SELect) or on the value of a floating point variable or expression (floating point SELect). Integer SELect is more efficient.

SBASIC checks for incorrectly nested or inconsistent SELect clauses.

WHEN ERRor

WHEN ERRor is suppressed within the command line to stop SBASIC rushing off into your error processing if you mistype a command.

You can turn off WHEN ERRor by executing an empty WHEN ERRor clause.

```
100 WHEN ERRor
110       CONTINUE                      :REMark  ignore errors
120 END WHEN
130 a = 1 / 0                           :REMark  no error
140 WHEN ERRor                          :REMark  restore error processing
150 END WHEN
160 a = 1 / 0                           :REMark  BANG!!
```

Loop Handling

FOR Loop Types

SuperBASIC requires FOR loops to have a floating point control variable. SBASIC allows both floating point and integer control variables. Integer FOR loops are more efficient than floating point for loops: particularly if the control variable is to be used to index an array.

```
FOR i% = 0 to maxd%: array(i%) = array(i%) * 2:     is preferred to

FOR i = 0 to maxd: array(i) = array(i) * 2:         which is less efficient
```

N.B. the type is determined before the program is executed.

In-Line Loops

Whereas SuperBASIC only allows a single structure to be defined "in-line", SBASIC allows many loops (and other structures) to be nested in-line without requiring END statements:

```
100  FOR i = 1 TO n: FOR j = 1 TO m: a(i,j) = a(i,j) + b(i,j)
```

The "NEXT Bug"

The "NEXT bug" reported in many articles about SuperBASIC, which many people have asked to be fixed, has not been fixed. IT IS NOT A BUG. NEXT is **defined** to fall through to the next statement when the loop is exhausted. It does not go to the statement after the END FOR (which may not be present). If that is what you wish to do, follow the NEXT by an EXIT.

Unnamed NEXT, EXIT and END Statements

Loop structures are "opened" with a FOR or REPeat statement and closed with an END FOR or END REPeat statement. SuperBASIC requires all loop closing statements as well as the intermediate NEXT and EXIT statements to identify the loop to which they apply. SBASIC, on the other hand, will accept unnamed NEXT, EXIT, END FOR and END REPeat statements. These are applied to the most recent (innermost) unclosed loop structure.

```
100  FOR i = 1 TO 10
110       FOR j = 1 TO 10
120            IF a(i,j) < 0: EXIT          implicitly EXIT j
130            sum = sum + a(i,j)
140       END FOR                           implicitly END FOR j, closes FOR j
150       IF sum < 100: NEXT                loop j is closed, so this is NEXT i
160       PRINT i,sum
170       sum=0
180  END FOR                                implicitly END FOR i, closes FOR i
```

REPeat Loops

Whereas SuperBASIC requires all REPeat clauses to have a name, SBASIC allows unnamed REPeats. These unnamed REPeats may be combined with with unnamed NEXT, EXIT and END REpeat statements.

```
100  REPeat
110       a$ = INKEY$(-1)
120       IF a$ = ESC$: EXIT                goes to 200 (outer loop)
130       IF a$ <> 'S': NEXT                goes to 110 (outer loop)
130       REPeat
140            a$ = INKEY$(-1)
150            IF a$ = ESC$: EXIT                goes to 180 (inner loop)
160            x$ = x$ & a$
170       END REPeat                             goes to 140 (inner loop)
180       IF LEN (x$) > 20: EXIT            goes to 200 (outer loop)
190  END REPeat                             goes to 110 (outer loop)
200 PRINT 'DONE'
```

Multiple Index Lists and String Slicing

For various reasons SBASIC does not support multiple index lists.

```
100 DIM a(10,10,10)
110 a(3,4)(5) = 345            OK for SuperBASIC, SBASIC will not handle this
120 a(3,4,5) = 345             Means the same, is easier to type and SBASIC likes it
```

To make up for this limitation, SBASIC allows you to slice strings at any point in an expression.

```
200 a$ = 2468 (3)              Sets a$ to '6' in SBASIC, prohibited in SuperBASIC
210 ax=1234
220 a$ = ('abcdef' & ax) (5 to 8)  Sets a$ to 'ef12' in SBASIC
230 b$ = 'abcdefghi'
240 a$ = b$(2 TO 7)(3 TO 5)(2)  Sets a$ to ''d' in either SBASIC or SuperBASIC
```

Also, in SBASIC, the default range for a string or element of a string array is always (1 TO LEN(string)) and zero length slices are accepted at both ends of a string (i.e. a$(1 to 0) or a$(LEN(string)+1 TO LEN(string)) are both null strings).

Writing Compiler Compatible Programs

SuperBASIC programs which are written in such a way as to be used both compiled and interpreted by SuperBASIC often have a small code fragment at the start to allow for the differences in compiled and interpreted environments.

The problem is not that SBASIC is "incompatible" with these code fragments but that SBASIC is compatible with SuperBASIC in a way which the two "compiled" SuperBASICs are not. The simplest way to avoid these problems is to give up using compiled BASIC and remove the junk from your programs. If, on the other hand, you wish to continue using compiled BASIC and also wish to use these programs in SBASIC daughter jobs, you may require some code changes.

There are three principal differences between the SuperBASIC environment and the Liberator and Turbo environments.

1. When executing in compiled form, the program will probably not be requiring windows #0, #1 and #2 in the same form as when it is being interpreted by SuperBASIC. In particular:

   - channel #0 (the command channel) may not be required at in the compiled version, but it is essential to keep it open in the SuperBASIC version otherwise no commands can ever be given again.;
   - a compiled program may be started with no windows open, a program interpreted by SuperBASIC will (usually) start with windows #0, #1 and #2 open.

   This distinction is not so much a difference between compiled and not compiled, but is a difference between interpreting a program within the permanent SuperBASIC interpreter and executing a transient program.

2. An interpreted program may be interrupted and rerun (so that the starting state may be different each time), while a compiled program will always start "clean" (always having the same starting state).

3. An interpreted program will report error messages to window #0 while compiled programs have their own error message facilities.

From the point of view of the last two differences, SBASIC is always much closer to SuperBASIC than to a compiled BASIC. For the first (and most important difference) SBASIC can behave either like a compiled BASIC or SuperBASIC.

- If SBASIC is started off with an SBASIC command, then SBASIC behaves like SuperBASIC: window #0 (at least) is open.
- If SBASIC is started off with an EX (etc.) command or from a HOTKEY or QPAC2 EXEC menu, then SBASIC behaves more like a compiled program: there are no windows open by default and window #0 is not required.

Unfortunately, the code that usually appears at the start of these compatible programs does not distinguish between compiled and interpreted environments, but between job 0 and other jobs.

```
100 IF JOB$(-1)<>''                     :REMark is it a named job (NOT SuperBASIC)
110      CLOSE #0,#2                    :REMark close spare windows in case
120      OPEN #1,con_512x256a0x0        :REMark our #1
130 ELSE
140      WINDOW 512,256,0,0             :REMark for SuperBASIC, just set #1
150 END IF
160 CLS
```

When used in an SBASIC daughter job, this will treat SBASIC as compiled whereas it should possibly be treated as interpreted as SBASIC programs can be re-run.

The problem cannot be resolved by using a function to distinguish between compiled, SuperBASIC and SBASIC, as there is no such function in SuperBASIC and it cannot be assumed that a suitable extension has been loaded.

SBASIC jobs are, however, always called SBASIC until the name is set by the JOB_NAME command.

The best approach would be to have program start up code which is sensitive to the environment and not having a different behaviour just because the job number is 0 or the job has no name. This is however, not practical with the old QL BASIC compilers.

The least bad solution may be to have a "four way switch" at the start of the program.

```
100 my$ = 'myjob': j$ = JOB$(-1)    :REMark  set my assumed and real names
110 IF j$ = ''                      :REMark  is it an unnamed job (SuperBASIC)?
120      do SuperBASIC or SBASIC job 0 fiddles
130 END IF
140 IF j$ = 'SBASIC'                :REMark  is it start of an SBASIC daughter?
150      do SBASIC daughter initialisation
160      job_name my$               :REMark  from now on it is a named job
170      j$ = ''                    :REMark  no further action required
180 END IF
190 IF j$ = my$                     :REMark  is it rerun an SBASIC daughter?
200      do SBASIC daughter re-initialisation
210      j$ = ''                    :REMark  no further action required
220 END IF
230 IF j$ <> ''                     :REMark  must be compiled!
240      do compiled BASIC initialisation
250 END IF
```

Within the initialisation code for SBASIC, the DEVTYPE function may be used to determine whether a channel is open.

This returns an integer value of which only the most significant (the sign bit) and least significant two bits are set. To ensure future compatibility, nothing should be assumed about the other bits.

The value returned will be negative if there is no channel open. Otherwise bit 0 indicates that it will support window operations (i.e. it is a screen device), bit 1 indicates that it will support file positioning operations (i.e. it is a file).

```
100 a% = DEVTYPE (#3)                 :REMark  find the type of device open as #3
110 IF a% < 0: PRINT '#3 not open'    :REMark  negative is not open
120 SELECT ON a% && %11               :REMark  ensure we only look at bits 0 and 1
130     = 0: PRINT '#3 is a purely serial device'
140     = 1: PRINT '#3 is a windowing device'
150     = 2: PRINT '#3 is a direct access (filing system) device'
160     = 3: PRINT '#3 is totally screwed up'
170 END SELECT
```

Error Reporting and Statement Numbering

SBASIC will, usually, report error in the form:

```
At line 250:3 end of file
```

The number after the colon is the statement number within the line.

N.B. SBASIC generates a small number of additional statements (jumps round DEF PROCs, jumps to END SELect before each ON and END statements on inline clauses) which are not visible in the SBASIC program. If you like piling up structures and statements into a single line, you may find that the statement number in the error report is larger than you would expect!

Extended SuperBASIC Commands and Functions

LOAD, LRUN, MERGE and MRUN

LOAD, LRUN, MERGE and MRUN have been extended to accept Liberation Software's _SAV file format. In addition, if the filename supplied is not found, SBASIC will try first with _BAS and then _SAV added to the end of the filename.

SAVE and SAVE_O

If no filename is given, the name of the file that was originally loaded will be used (if necessary substituting _BAS for _SAV at the end). The file will be saved with a version number one higher that the file version when it was LOADed. (Repeated SAVEs do not, therefore, keep on incrementing the version number).

If a filename is given, the version number is sert to 1.

QLOAD and QLRUN

The extension of the SBASIC LOAD command makes the real QLOAD and QLRUN commands (which require a copy or near copy of QDOS ROMs to function at all) nearly redundant. QLOAD and QLRUN are implemented in SBASIC as versions of LOAD and LRUN that ensure that there is a _SAV at the end of the filename.

QMERGE, QMRUN, QSAVE and QSAVE_O

These are versions of MERGE, MRUN, SAVE and SAVE_O which work with _SAV files.

If there are 4 SBASIC programs in the data default directory in called FRED, JOE, ANNE and CLARA with either _BAS or _SAV at the end of the names.

```
FRED
JOE_BAS
ANNE_SAV
CLARA_BAS
CLARA_SAV
```

QLOAD fred	*Fails as there is no FRED_SAV*
LOAD fred	*Loads FRED*
SAVE	*Saves the program as FRED*
QSAVE	*Saves the program as FRED_SAV (quickload format)*
SAVE junk_bas	*Saves the program as JUNK_BAS*
QSAVE	*Saves the program as JUNK_SAV (quickload format)*
MERGE joe	*Merges the file JOE_BAS into the program*
MERGE anne	*Quick merges the file ANNE_SAV into the program*
SAVE	*Saves it as JUNK_BAS (MERGE does not change the name)*
LOAD clara_bas	*Loads CLARA_BAS*
QLOAD clara	*Quick loads CLARA_SAV*
LOAD clara_sav	*Also quick loads CLARA_SAV*

LBYTES
SBYTES SBYTES_O
SEXEC SEXEC_O

All accept a channel number in place of a name. This can improve efficiency.

```
nc = FOPIN ('file')             Open file once only
base = ALCHP (FLEN(#nc))            . . . to allocate bit of heap
fdt = FUPDT (#nc)               . . . get the update date
LBYTES #nc,base                     . . . and load it
CLOSE #nc
```

EPROM_LOAD

The EPROM_LOAD (*filename*) command is a special trick for loading the image of a QL EPROM cartridge. Most EPROM cartridges are programmed so that the cartridge may be at any address. Some require to be at exactly $C000, the QL ROM port address. The first time the command is used after reset, the EPROM image will be loaded at address $C000. Subsequent images may be loaded at any address. Fussy EPROM images must, therefore, be loaded first. An EPROM image file must not be longer than 16 kilobytes.

To make an EPROM image, put the EPROM cartridge (for example the Prospero PRL cartridge) into your QL and turn on. SBYTES the image to a suitable file with the magic numbers 49152 ($C000) for the base address and 16384 (16 kilobytes) for the length.

```
SBYTES flp1_prl, 49152, 16384  Save Prospero PRL image
```

On your SMSQ machine copy the file to your boot diskette or disk and add the EPROM_LOAD statement to your "boot" file.

```
EPROM_LOAD flp1_prl            Load Prospero PRL image
```

QL Based Hardware

SMSQ on QL based hardware recognises plug-in ROM cartridges and copies them fast memory when the system is booted. EPROM_LOAD can still be used, however, to load ROM images. If the ROM slot is vacant, then the first EPROM load will load to the QL ROM Port address. Otherwise, all EPROM images will be loaded to arbitrary addresses.

PEEK$ POKE$

PEEK$ (*address, number of bytes*) returns a string with the number of bytes starting from address. The bytes need not, of course, be text.

POKE$ (*address, string*) pokes the bytes of the string starting from the address.

PEEK$ and POKE$ can be used for copying memory.

```
a$ = peek$ (base1,1000)     Peek 1000 bytes from address base1
poke$ base2,a$                  . . . and poke them back to base2
```

PEEK$ and POKE$ can accept all the extended addressing facilities of PEEK and POKE. Indeed, POKE$ is identical to POKE which can now accept string parameters.

PEEK PEEK_W PEEK_L
POKE POKE_W POKE_L

The standard PEEK functions and POKE procedures have been extended to provide compatibility with the Minerva versions. There are three main changes.

1. The address may be specified relative to the base of the system variables or the (current) SBASIC variables.
2. The contents of the memory at the address may itself be used as a base address with a second value providing an offset for this address.
3. More than one value may be POKEd at a time.
    - For POKE_W and POKE_L, the address may be followed by a number of values to poke in succession.
    - For POKE the address may be followed by a number of values to poke in succession and the list of values may include strings. If a string is given, all the bytes in the string are POKEd in order. The length is not POKEd.

Absolute PEEK, POKE

The standard forms of PEEK and POKE are supported even though the use of PEEK and POKE is best regarded as a form of terrorism.

a=RESPR (2000)	
LBYTES myfile,a	*Load myfile*
PRINT PEEK (a),	*Prints the value of the byte of myfile*
POKE_L a+28, DATE,0	*Set the 28th to 35th bytes to the DATE (4 bytes) and 4 zeros*
POKE a+8, 0,6,'My_Job'	*Set the standard string (word length followed by the chars)*

Peeking and Poking in the System Variables

If the first parameter of the peek or poke is preceded by an exclamation mark, then the address of the peek or poke is in the system variables or referenced via the system variables. There are two variations: direct and indirect references.

- For direct references, the exclamation mark is followed by another exclamation mark and a an offset within the system variables.
- For indirect references, the exclamation mark is followed by the offset of a pointer within the system variables, another exclamation mark and an offset from that pointer.

ramt = PEEK_L (!!$20)	*Find the top of RAM $20 bytes on from the base of sysvars*
POKE_W !!$8e,3	*Set the auto-repeat speed to 3*
job1 = PEEK_L (!$68!4)	*Find the base address of Job 1 (4 on from base of Job table)*
POKE !$B0!2, 'WIN'	*change the first three characters of DATA_USE to WIN*

There is slightly more parameter checking than in the Minerva versions. Nevertheless, errors and deliberate abuse are not likely to be detected and may have different effects on SMSQ and Minerva.

Peeking and Poking in the SBASIC Variables

If the first parameter of the peek or poke is preceded by an backslash, then the address of the peek or poke is in the SBASIC variables or referenced via the SBASIC variables. There are two variations: direct and indirect references.

- For direct references, the backslash is followed by another backslash and an offset within the SBASIC variables.
- For indirect references, the backslash is followed by the offset of a pointer within the SBASIC variables, another backslash and an offset from that pointer.

```
dal = PEEK_W (\\$94)          Find the current data line number

n6 = PEEK_W (\$18\2+6*8)  Find the name pointer for the 6th name in the name table
nl6 = PEEK (\$20\n6)              . . . and the length of the name
n6$ = PEEK$ (\$20\n6+1, nl6)      . . . and the name itself.
```

BPUT BGET

BPUT will accept string parameters to put multiple bytes. BGET will accept a parameter that is a sub-string of a string array to get multiple bytes.

```
BPUT #3,27,'R1'                      Put ESC R 1 to channel #3
DIM a$(10): BGET #3, a$(1 to 6)      Get 6 bytes from #3 into a$
```

ATAN

The ATAN function has been extended to provide 4 quadrant result by taking two parameters. If x is greater than 0, ATAN (x,y) give the same results as ATAN (y/x). Otherwise it returns values in the other quadrants ($>\pi/2$ and $<-\pi/2$).

PROT_DATE

Where the system has a separate battery backed real time clock. The date is read from the clock when the system is reset. Thereafter, the clock is kept up to date by the SMSQ timer. (Thus the impressive speed gains made by some accelerator software: slowing the clock down by disabling interrupts can do wonders for your benchmark timings).

In general, the system real time clock is updated whenever you adjust or set the date. As some QL software writers could not resist the tempation of setting the date to their birthday (or other inconvenient date) this can play havok with your file date stamps etc.

PROT_DATE (*0 or 1*) is used to protect (1) or unprotect (0) the real time clock. If the real time clock is protected, setting the date affects only SMSQ's own clock, the real time will be restored then next time the computer is reset.

```
PROT_DATE 1          protect the RTC (should never be required)
PROT_DATE 0          unprotect the RTC (normal)
```

VER$

The VER$ function has been extended to take an (optional, Minerva compatible) parameter. If it is non zero, information is taken from the OS call for system information. Otherwise, the normal SBASIC version (HBx) is returned.

PRINT ver$	*prints HBA (or later SBASIC version ID)*
PRINT ver$(0)	*also prints HBA (or later SBASIC version ID)*
PRINT ver$(1)	*prints 2.22 (or later SMSQ version number)*

With a negative parameter, VER$ does not return a version at all, but returns a fairly arbitrary choice of information.

PRINT ver$(-1)	*prints the Job ID (0 for initial SBASIC)*
PRINT ver$(-2)	*prints the address of the system variables (163840), WHY?*

Multiple Copies of SBASIC

There never was much problem getting multiple copies of SuperBASIC to run under QDOS. There is even less of a problem getting multiple copies of SBASIC to run under SMS. The problem was always what to do with the windows.

SBASIC has four distinct forms.

1. Job 0 is the "guardian" of SBASIC extensions, permanent memory allocation and channel 0.
2. SBASIC "daughter jobs" may be created with the SBASIC command. These may be created with the same set of 3 windows as the initial Job 0 windows. Alternatively, they may be created with a single channel #0 or even no windows open at all.
3. SBASIC source files (ending in _bas) may be executed by EX, EXEC, EW, EXEC_W.
4. SBASIC may be invoked as a Thing which may either operate within the context of an invoking Job, or, once set up, operate as an independent daughter Job.

SBASIC Daughter Jobs

Having a number of SBASIC jobs which completely cover each other may not be very useful. SBASIC daughter jobs may, therefore, either be created either with the full set of standard windows (in which case they all overlap) or they may be created with only one small window (#0).

The SBASIC command, which creates SBASIC daughter jobs, has an optional parameter: the x and y positions of window #0 in a one or two digit number (or string).

- If no parameter is given, the full set of standard windows will be opened.
- Otherwise, only window #0 will be opened: 6 rows high and 42 mode 4 characters wide within a 1 pixel wide border (total 62x256 pixels).
  - If only one digit is given, this is the SBASIC "row" number: row 0 is at the top, row 1 starts at screen line 64, row 4 is just below the standard window #0.
  - If two digits are given, this is the SBASIC "column, row" (x,y) position: column 0 is at the left, column 1 starts at 256 pixel in from the left.

SBASIC	*create an SBASIC daughter with the 3 standard windows*
SBASIC 1	*create an SBASIC daughter with just channel #0 in row 1*
SBASIC 24	*create an SBASIC daughter to the right of and below the standard windows (a 800x600 display is required)*

Because it is quite normal for an SBASIC job to have only #0 open, all the standard commands which default to window #1 (PRINT, CLS etc.) or window #2 (ED, LIST etc.) will default to window #0 if channel #1 or channel #2 is not open. This may not apply to extension commands.

If you have a screen larger than 512x256 pixels, it is useful to be able to re-position the SBASIC windows. The TK2 WMON and WTV commands have been extended to take an extra pair of parameters: the pixel position of the top left hand corner of the windows. If only one extra parameter is given, this is taken to be both the x and y pixel positions.

WMON 4,50	*reset windows to standard monitor layout displaced 50 pixels to the right and 50 pixels down.*

If the mode is omitted, the mode is not changed, and, if possible, the contents are preserved and the outline (if defined) is moved.

WMON ,80.40	*reset windows to standard monitor layout displaced 80 pixels to the right and 40 pixels down, preserving the contents*

A border has been added to window #0 to make it clearer where an SBASIC Job is on the screen.

JOB_NAME

The procedure JOB_NAME (*job name*) can be used to give a name to an SBASIC Job. It may appear anywhere within a program and may be used to reset the name whenever required. This command has no effect on compiled BASIC programs or Job 0.

JOB_NAME Killer	*sets the Job name to "Killer"*
JOB_NAME "My little Job"	*sets the Job name to "My little Job"*

Executing SBASIC Programs

SBASIC program files (ending in _BAS, _bas, _SAV or _sav) may be executed using the EX (EXEC) and EW (EXEC_W) commands.

EX my_little_prog_bas	*executes the SBASIC program "my_little_prog_bas"*

Just as for "executable" programs, if file or device names (or channels) are given after the program name, the first file device or channel will be #0 within the program, the second will be #1 etc.

A simple program for "uppercasing" could be

```
100  JOB_NAME UC
110  REPeat
120        IF EOF(#0): QUIT
130        BGET #0,a%
140        SELect ON a% = 97 to 122: BPUT #1, a% ^^ 32: = REMAINDER BPUT #1, a%
150  END REPeat
```

Saved as "uc_bas", this can be used for printing a file in upper case:

```
EX uc_bas, any_file, par
```

It can also be used as a filter to uppercase the output of any program sending its output to the "standard output".

```
EX my_prog TO uc_bas, par
```

The command QUIT should be used to get rid of an SBASIC job whether it has been created by the SBASIC command, EX or any other means.

Channel #0

There are some oddities in the handling of channel #0 which have been introduced to make the use of SBASIC a little easier.

- On normal completion of a program, if #0 is not open. SBASIC will die naturally. If #0 is open, SBASIC will wait for a command.
- In case of error, if #0 is not open, a default window #0 will be opened for the error message.
- Likewise, if an operation is requested on a default channel (#0, #1 or #2) and neither the default channel nor #0 are open, a default window #0 will be opened for the operation.

SBASIC and Resident Extensions.

Resident extensions linked into Job 0 (the initial SBASIC) are available to all SBASIC jobs. If extension procedures and functions are linked into other SBASIC Jobs (using LRESPR), they are local to those Jobs and will be removed when the Jobs die or are removed.

Note that, because of this feature, LRESPR cannot be used from a Job, other than Job 0, to load files which include system extensions (i.e. MENU_REXT, QTYP etc.).

SBASIC Executable Thing

The SBASIC executable Thing is called "SBASIC". The provision of an SBASIC executable Thing enables the diehard QDOS fanatic to go well beyond the facilities provided by the SBASIC and EX commands. Depending on how it is invoked, SBASIC can execute independantly of the invoking program, or it may take its channels and program from the invoking program.

On being invoked, SBASIC expects to find some channel IDs and a string on the stack (standard QDOS conventions). Because, however, SBASIC requires some BASIC source code in order to be able to execute, the treatment these channel IDs and the string on the stack are slightly unconventional.

- If SBASIC is invoked without any channel IDs on the stack, SBASIC will behave either as a normal SBASIC interpreter, with the standard set of windows, or as an interpreter with no windows initially opened.
  - If the string on the stack is null, the standard set of windows is opened and SBASIC waits for a command.

    (This is what happens when you give an SBASIC command without parameters or when you start SBASIC from the QPAC2 EXEC menu without a command string.)
  - If the string on the stack is not null, no windows are opened and the string is treated as a command line.

    (This is what happens when you start SBASIC from the QPAC2 EXEC menu after specifying a command string.)
- If there are one or more channel IDs on the stack, SBASIC will normally treat the first ID as the SBASIC program source file, the next ID as channel #0, the next ID as channel #1 and so on. The string defines the initial value of the cmd$ variable within the SBASIC program.

  (This is what happens when EX executes an SBASIC program.)

- There is a special "trick" for setting up an SBASIC program with just window #0 open. The x,y coordinates of the top left hand corner of the required window #0 are complemented and put on the stack in place of a channel ID.
    - If there is only one channel ID on the stack, and this is a "false" ID (i.e. the ID is negative), a 6 line by 42 column channel #0 is opened with the origin at NOT the MSW (x) and NOT the LSW (y) of the false ID. The string is treated as a command line.

      (This is what happens when you give a SBASIC command specifying the position of window #0.)
    - If there are two channel IDs on the stack and the second is a "false" ID, first is used as the SBASIC program source file and the second ID is used to define window #0. The string defines the initial value of the cmd$ variable.

      (This could be useful.)

The SBASIC Interface Things

Two interface Things are provided for the interface to the SBASIC extension which are compatible with two of the established executable program interfaces. The first is called "SBAS/QD" which provides a QD5 compatible F10 interface (Jochen Merz). The second is a "FileInfo" Thing (Wolgang Lenerz) which recognises and executes files starting with _sav or _bas.

If QD (version 5 or later) is configured to use the SBAS/QD thing, then you can create (line numbered or unnumbered) SBASIC programs with QD and execute them by pressing F10 (shift F5). QD may be temporarily configured to do this by executing it with the appropriate command string.

```
EX QD;'\T SBAS/QD'          Execute QD using SBAS/QD Thing
```

The FileInfo Thing is used by the QPAC2 Files Menu (amongst others) to determine how to "Execute" a file. With the default FileInfo Thing incorporated into SMSQ, files ending with _sav or _bas may be executed directly from the Files menu and any other utility program which uses the FileInfo Thing.

Input Line Editing

The range of standard input line editing keystrokes is now much wider. and has been made consistent for INPUT and ED.

Key	With	Operation
←		move left one character
→		move right one character
TAB	SHIFT	move left eight characters
TAB		move right eight characters
←	SHIFT	move left one word
→	SHIFT	move right one word
←	ALT	move to start of line
→	ALT	move to end of line
←	CTRL	delete left one character
→	CTRL	delete right one character
←	CTRL SHIFT	delete left one word
→	CTRL SHIFT	delete right one word
←	CTRL ALT	delete to start of line
→	CTRL ALT	delete to end of line
↓	CTRL	delete whole line

Some keyboards have Delete and Backspace keys.

Bspce		delete left one character
Delete		delete right one character
Bspce	SHIFT	delete left one word
Delete	SHIFT	delete right one word
Bspce	ALT	delete to start of line
Delete	ALT	delete to end of line

Language Facilities

SMSQ/QXL incorporates several language variations and extra variations may be add "at run time".

Language Specification

A language may be specified either by an international dialling code or an international car registration code. These codes may be modified by the addition of a digit where a country has more than one language.

Language Code	Car Registration	Language and Country
33	F	French (in France)
44	GB	English (in England)
49	D	German (in Germany)

Language Control Procedures

There is a set of procedures and functions which allow the language of the messages, the keyboard layout and the printer translate tables to be set. Where a language is to be specified, the parameter may be an integer value (the telephone dialing code), a string (the car registration letters) a variable or expression which yields an integer or string result, or a variable name.

It is not necessary for the car registration letters to be in upper case.

LANG_USE

The language of the messages is set by the LANG_USE *lang* command. This sets the OS language word, and then scans the language dependent module list selecting modules and filling in the message table.

LANGUAGE 33	*set language to French*
LANGUAGE D	*set language to German*
LANGUAGE 'g'&'b'	*set language to English*

WARNING: if you assign a value to a variable, then you will not be able to use that variable name to specify the car registration letters.

D=33: LANGUAGE D	*set language to French (dialing code 33) rather than German (car registration D)*

LANGUAGE LANGUAGE$

The LANGUAGE and LANGUAGE$ functions are used to find the currently set language, or to find the language that would be used if a particular language were requested. They can also be used to convert the language (dialing code) into car registration and vice versa.

PRINT LANGUAGE	*the current language*
PRINT LANGUAGE$	*the car registration of the current language*
PRINT LANGUAGE (F)	*the language corresponding to F*
PRINT LANGUAGE$ (45)	*the car registration corresponding to 45*
PRINT LANGUAGE (977)	*the language that would be used for Nepal*

KBD_TABLE

The keyboard tables are selected by the KBD_TABLE *lang* command.

KBD_TABLE GB	*keyboard table set to English*
KBD_TABLE 33	*keyboard table set to French*

Private keyboard tables may also be loaded.

i = RESPR (512): LBYTES "kt",i: KBD_TABLE i	*keyboard table set to table in "kt"*

For compatibility with older drivers, a "private" keyboard table loaded in this way should not be prefaced by flag word.

TRA

The SBASIC TRA command differs very slightly in use from the QL JS and MG TRA. The differences are quite deliberate and have been made to avoid the unfortunate interactions between functions of setting the OS message table and setting the printer translate tables. If you only wish to set the printer translate tables, the only difference is that TRA 0 and TRA 1 merely activate and disactivate the translate. **They do not smash the pointer to the translate tables if you have previously set it with a TRA *address* command**.

If you wish to change the system message tables, then the best way is to introduce a new language: this is done by LRESPRing suitable message tables.

Language dependent printer translate tables are selected by the TRA 1,*lang* command. If no language code or car registration code is given, the currently defined language is used.

Language independent translate tables are set by the TRA *n* command where *n* is a small odd number.

Private translate tables are set by the TRA *addr* command where *addr* is the address of a table with the special language code $4AFB.

TRA 0	*translate off, table unchanged*
TRA 0, 44	*translate off, table set to English*
TRA 0, F	*translate off, table set to French*
TRA 1	*translate on, table unchanged*
TRA 1, GB	*translate on, table set to English*
TRA 1, 33	*translate on, table set to French*
TRA 3	*translate on, table set to IBM graphics*
TRA 5	*translate on, table set to GEM VDI*
A = RESPR (512): LBYTES "tratab",A: TRA A	*translate on, table set to table in "tratab"*

To use the language independent tables, your printer should be set to USA (to ensure that you have all the # $ @ [ ] { } \ | ^ ~ symbols which tend to go missing if you use one of the special country codes (thank you ANSI)), and select IBM graphics or GEM character codes as appropriate.

For the IBM tables, QDOS codes $C0 to $DF are passed through directly and QDOS codes $E0 to $EF are translated to $B0 to $BF to give you all the graphic characters in the range $B0 to $DF. QDOS codes $F0 to $FF are passed though directly to give access to the odd characters at the top of the IBM set.

For the GEM tables, QDOS codes $C0 to $FF are passed through directly.

BAUD Command

For the QXL, the standard BAUD command mimics the QL BAUD command.

BAUD 4800	*Set SER1 and SER2 to 4800 baud*

Both the SuperBASIC BAUD command and the OS baud trap have been extended to support independent baud rates for each serial port.

BAUD 1,19200	*Set SER1 to 19200 baud*
BAUD 2,0	*Set SER2 to 153600 baud*

Virtual Devices

Virtual devices are not associated with any physical hardware. NUL devices are complete dummy (very useful for benchmarking: SMSQ/QXL has one of the fastest, if not the fastest, fully functional NUL device in the world). PIPEs 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).

NUL Device

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

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

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

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

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

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

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

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

PIPE Device

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

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

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

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

Named pipes should be opened with OPEN_NEW (FOP_NEW) for output and OPEN_IN (FOP_IN) for input. A named pipe is created when there is an open call for a named pipe which does not exist. It goes away when there are no longer any channels open to it **and** it is has been emptied.

As well as the name, it is possible to specify a length for a named pipe. If the pipe already exists, the length requested is ignored.

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

DEV - A Virtual Filing System Device

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

Each DEV (DEV1 to DEV8) device is a pseudonym for a real filing system device or directory on a filing system device.

Files on a DEV device can be OPENed used and DELETEd in the same way as tey can on the real device.

DEV_USE

Each DEV device is defined using the DEV_USE (*number, name, next*) which specifies the number of the DEV device, the real device or directory and the next device in the chain.

DEV_USE 1, ram1_	*DEV1_ is equivalent to ram1_*
OPEN #3, dev1_f1	*opens ram1_f1*
DEV_USE 2, flp1_ex_	*DEV2_ is equivalent to flp1_ex_*
OPEN #3, dev2_f1	*opens flp1_ex_f1*
DEV_USE 3, win1_work_new	*DEV3_ is equivalent to win1_work_new*
OPEN #3, dev3_f1	*opens win1_work_newf1*
DELETE dev3__junk	*deletes win1_work_new_junk*

Note that, unlike the defaulting commands PROG_USE and DATA_USE, the underscore at the end of the real device or directory is significant.

There is a neat variation on the DEV_USE call which enables you to to set up default chains. If you put a "next" number at the end of the DEV_USE command, this will be taken as the DEV to try if the open fails. This next DEV can also chain to another DEV. You can even close the chain: the DEV driver will stop chaining when it has tried all the DEVs in the chain.

DEV_USE 1, ram1_, 3	*DEV1_ is equivalent to ram1_, next is DEV3*	
DEV_USE 2, flp1_ex_, 1	*DEV2_ is equivalent to flp1_ex_, next is DEV1*	
DEV_USE 3, win1_work_, 2	*DEV3_ is equivalent to win1_work_ next is DEV2*	
LOAD dev1_anne	*will try*	*ram1_anne (DEV1)*
	then	*win1_work_anne (DEV3)*
	and finally	*flp1_ex_anne (DEV2)*

LOAD dev2_anne	*will try*	*flp1_ex_anne (DEV2)*
	then	*ram1_anne (DEV1)*
	and finally	*win1_work_ann (DEV3)*

Note that DELETE only operates on the DEV specified: it does not chain.

A DEV default may be cleared by giving no name.

DEV_USE 2	*clear definition for DEV2*

DEV_LIST

DEV_LIST (*channel*) lists the currently defined DEVs in the specified channel (default #1)

DEV_LIST	*lists the current DEVs in #1*
DEV_LIST#2	*lists the current DEVs in #2*

DEV_USE$ DEV_NEXT

The DEV_USE$ (*number*) function returns the usage for the specified DEV. The DEV_NEXT (*number*) function returns the next DEV after the specified DEV.

PRINT DEV_USE$(3)	*prints the usage for DEV3*
PRINT DEV_NEXT(1)	*prints the next DEV in the chain after DEV1*

Interaction between DATA_USE, PROG_USE and DEV.

If you are going to use the DEV defaults, it makes sense to set the DATA_USE and PROG_USE defaults to use DEV, and when moving from directory to directory change the DEV definition rather than the DATA_USE.

DATA_USE dev1_	data default directory is DEV1_
DEV_USE 1, flp2_myprogs_	. . . which is myprogs on FLP2
PROG_USE dev2_	programs from DEV2_
DEV_USE 2, flp1_ex_, 1	. . . which is flp1_ex or my data default!