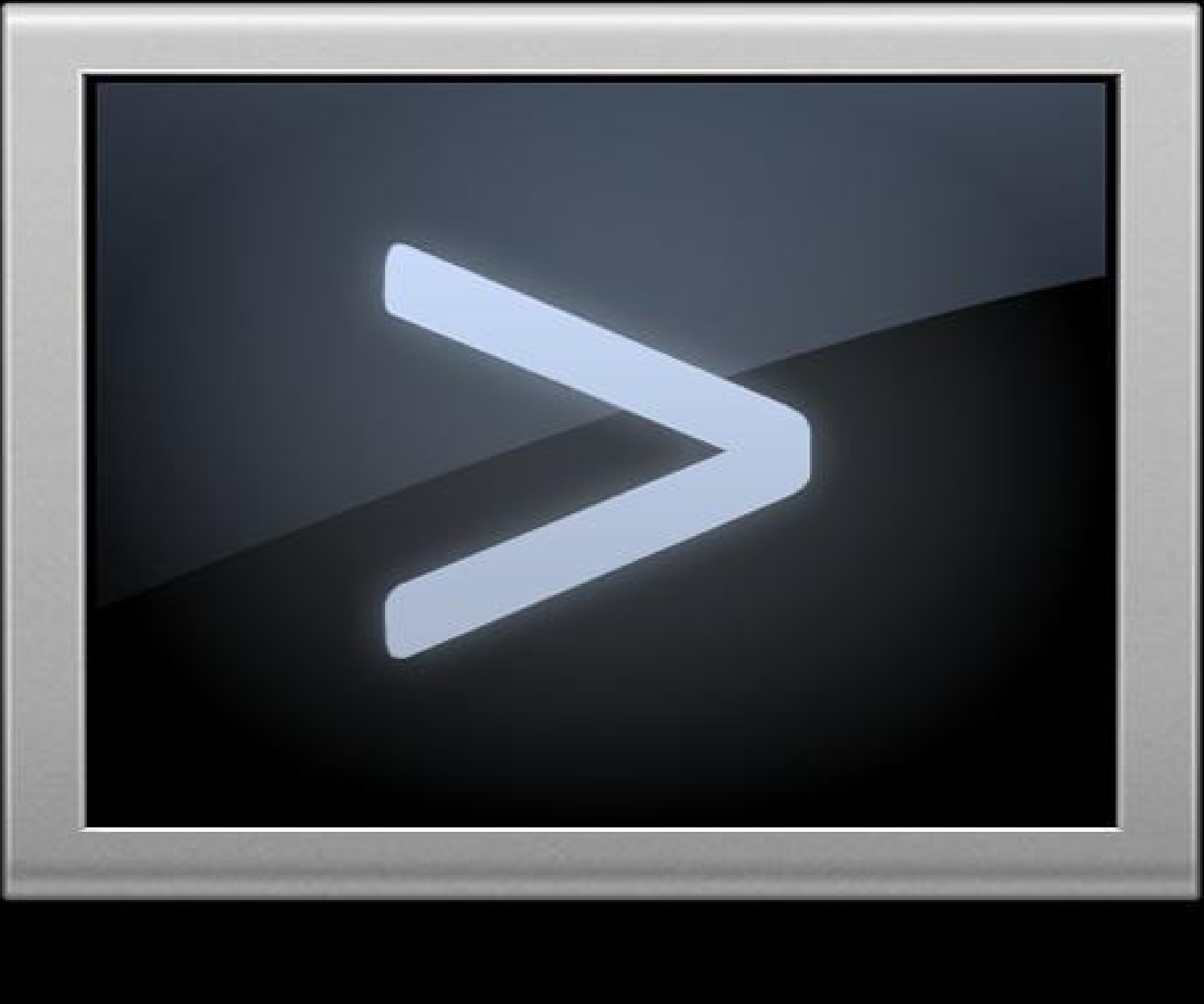
QL SHELL USER GUIDE



F1...monitor
F2...TV

QL Shell User Manual

A simple UNIX-style Command Shell for QDOS and SMS

v1.12 by Adrian Ives, Memory Lane Computing Ltd, 06-JAN-2012

(Note: Original Shell project by P.J.Taylor, 08-NOV-1991)

Introduction

The QL Shell is an interactive command processor to support the control and execution of Jobs under QDOS and SMSQ. Its primary purpose is to deliver a quick interface to the Operating System job control facilities to help developers using compiled languages such as C. Its secondary purpose is to deliver a simple scripting facility for automating sequences of Jobs.

Running QL Shell

QL Shell is executed like any other QDOS/SMSQ program, and if you use the Toolkit II EX/EW commands to start it then you have the advantage that you can pass channels and/or a command string.

```
    EW 'Sh',#1
```

Will take input from S*BASIC Channel #1 and also send its output there.

```
    EW 'Sh',#0,#1
```

Will take input from S*BASIC Channel #0 and echo completed command lines to S*BASIC Channel #1, together with the output. Similarly:

```
    EW 'Sh',#0,#2,#1
```

Will take Standard Input from #0, and send Standard Output to #1 and Standard Error to #2 (note the order '#0,#2,#1' this follows TK II and C68 conventions).

A parameter string may also be supplied to redirect standard input and standard output, as usual for all C68 compiled programs, using the '>' and '<'. symbols.

When Standard Input and Standard Output are both directed to the same console device each command line is prompted for using a '>'. There are two main types of commands that QL Shell can process: External and Internal. The command itself is always case-independent, but parameters are passed to external programs with no character translation.

The autoexec_bat/autoshell_bat file

When QL Shell is initially loaded it will look for a file called *autoexec_bat* in the current Data Directory. If it finds this file, then it will execute all of the commands in it. If autoexec_bat is **not** found, then it looks for *autoshell_bat* and executes that instead. This is useful for setting up your favourite settings at the start of a session, and is particularly convenient for setting aliases. You can override the name of this file by setting the environment variable SHELL_AUTO to the full name of the file that you wish to use instead.

Forcing the execution of a command

If the command line passed to QL Shell begins with an @ sign, the remainder of the line is assumed to be a command for the Shell to execute immediately. On completion of executing such a command, the Shell will then exit.

```
    EW 'Sh',#1;'@ls'
```

This (pointless) example will load QL Shell, which will then load the external LS command, sending the output to channel #1 (this is exactly the same as doing `EX 'ls',#1` but it does show the principle).

This facility is somewhat limited. In particular any redirection symbols that you include are taken as applying to the Shell, and not to the command. This is because of the way in which the C68 run-time system processes the command line for maximum compatibility with UNIX systems. Please bear this in mind if using the '@' option.

Because of this use of the '@' sign you should regard this symbol as a reserved character, and avoid using it in parameters passed to batch files - this could cause the Shell some confusion.

Special Symbols

QL Shell recognises a number of special symbols, the values of which are substituted **before** any command line is processed. All special symbols begin with a dollar sign ($). To use a single $ sign anywhere in a command, use two of them ($$) instead.

Symbol	Meaning

	Standard Symbols
	The standard symbols gain access to Batch File parameters and the current values of QL Shell's looping constructs.
$.	The number of parameters that were passed to the batch file.
$A..$Z	The values of the first 26 parameters that are supplied to a batch file. e.g: `EX 'Sh';'Hello World'` When the commands in the file MyBatch_bat are executed, $A will be the string "Hello", and $B will hold the string "World".
$*	The current filename in a FOR command (and only returns a value in that context)
$/	As for $*, but returns only the filename (i.e: The directory is stripped off the front).
$^	The current filename in a FOR command, less its extension. (Also only returns a value when used in a FOR command).
$\	As for $^, but returns only the filename (i.e. The directory is stripped off the front, as well as the extension off the end).
$+	The current loop counter value in a DO command (only returns a value when actually referenced within a DO command).
$=	The result of the last FTEST or DTEST command. This is a shorthand way of accessing the $(%result) variable.
$0..$9	Within an alias, or a batch file that has been CALLed, the values of the first ten parameters. Outside of this context, no value is returned.
$-	Within an alias, or a batch file that has been CALLed, the complete set of parameters concatenated together, and separated by single spaces. Outside of this context, no value is returned.
$:	Within an alias, or a batch file that has been CALLed, returns the total number of parameters that were supplied.
$(Name)	The value of the environment variable *Name* (Although you do not need environment variable support loaded for this to work, because C68 programs maintain their local environment which will be passed down to child processes, so many other QL utilities now use environment variables that you should seriously consider including Env_Bin in your boot program).
	Shell Internal Variables
	The syntax $(%Name) returns the value of one of QL Shell's Internal Variables.
%ver	The current Shell version number
%data	The current Data Directory
%prog	The current Program Directory
%dest	The current Destination Directory
%osdata	The current Data Directory, as maintained by QDOS/SMSQ
%osprog	The current Program Directory, as maintained by QDOS/SMSQ
%osdest	The current Destination Directory, as maintained by QDOS/SMSQ
%retcode	The return code from the last job executed
%result	The result of the last FTEST or DTEST command. You can also access this value with the more convenient shorthand $= symbol.
%lastkey	The ASCII code of the key that was pressed to exit from a PAUSE
%date	The current date in the form: Day Month Date Year
%time	The current time in the form: HH:MM:SS
%longdate	The current date and time in full format: Weekday Month Day hh:mm:ss YYYY
%day	Current day (of the month - i.e. a number)
%month	Current month as characters (jan, feb, mar etc.)
%year	Current year
%weekday	Current day (of the week) as characters (sun, mon, tue etc.)
%hour	Current hour
%minute	Current minute
%second	Current second
	Medium Information Functions
	These all return information about the medium that the Data Default directory is set to point to.
%name	Medium name
%alloc	Allocation Unit size (bytes)
%free	Free space (in Allocation Units)
%kfree	Free space (in KB)
%mfree	Free space (in MB)
%total	Total space (in Allocation Units)
%ktotal	Total space (in KB)
%mtotal	Total space (in MB)

	Note: If any of these functions fail (e.g. not a Level 2 device driver), the string E*-n is returned, where -n is the QDOS error code.
	String Functions
	Syntax: $(%func params:operand) This special syntax is used to introduce some simple string handling functions to manipulate values (normally filenames). The operand may be one of the following:- %name = An Internal Variable. $A ..$Z and $. = Batch File parameters. $* $^ $/ $\ $+ and $. = The looping variables. name = An Environment Variable. (If it cannot be found, *name* is used as a literal string).
	Note: Function calls are evaluated left to right so it is possible to do some rudimentary nesting by using the result of a function as the operand to the next. What you cannot do is to try and use the result of one function as a parameter to another. (Note the distinction between the parameters and the operand. The operand is acted upon by the function according to the parameters). So: `$(%rev:%mid 2,2:%ver)` works (and gets 70 for v1.07!), but:`$(%rev:%mid 2,%second:AnEnvironmentVar)` does not!
	List of Functions
%uc	Shift the operand to upper case. e.g: `$(%uc:WIN2_MyFile)` = WIN2_MYFILE
%lc	Shift the operand to lower case. e.g: `$(%lc:WIN2_MyFile)` = win2_myfile
%chr	Returns the single character whose ASCII code is specified as the argument. e.g: `$(%chr:65)` = A
%rev	Reverse the operand. e.g: `$(%rev:Hello World)` = dlroW olleH
%dname	Return the directory part of a full filename. Only works if the file is present on a true Level 2 device driver that uses file type 255 for directories. e.g: If there is a file called sh_c in WIN2_C_SH, then `$(%dname:WIN2_C_SH_sh_c)` returns the string WIN2_C_SH.
%fname	Return the name part of a full filename. Only works if the file is present on a true Level 2 device driver that uses file type 255 for directories. e.g: If there is a file called sh_c in WIN2_C_SH, then `$(%fname:WIN2_C_SH_sh_c)` returns the string sh_c
%part n	Return the nth, underscore-delimited, part of the filename. Use a negative value to extract a part counting from the right. e.g: `$(%part 4:WIN2_C_SH_sh_c)` and `$(%part -2:WIN2_C_SH_sh_c)` both = sh
%left n	Return n characters from the left of the operand. e.g: `$(%left 5:WIN2_Hello_exe)` = WIN2_
%right n	Return n characters from the right of the operand. e.g: `$(%left 3:WIN2_Hello_exe)` = exe
%mid n,m	Return m characters from position n of the operand (n starts at 0). e.g: `$(%mid 5,5:WIN2_Hello_exe)` = Hello
%len	Return the length of the operand. e.g: `$(%len:hello)` = 5 (Assuming that there was no Environment Variable with the name *hello*!)
%err	Return the text of the error message associated with the QDOS error number specified by the operand. e.g: `$(%err:-19)` = Not implemented
	Note: Parameter checking for these functions is minimal. If you supply illegal values the end result is usually (but not always) an empty string. The Shell's internal buffers are 128 bytes. This is fine for manipulating QDOS filenames that cannot exceed 42 characters. But, if an element of a string substitution exceeds 128 bytes you know what the result will be!
	File Functions
	Syntax: `$(%func params:operand)` File functions work exactly like the string functions mentioned above, except that they take a filename as their operand. Normal directory search rules will be applied to find the file. (i.e. the Data Default is searched). All file functions either return the value requested, or the string E*n, where n is a negative QDOS error code.
%isdir	Returns 1 if the operand represents a valid directory, or a valid directory device (in which case, do NOT include the trailing underscore). Otherwise returns 0. Note that this function behaves exactly like the DTEST command, except that it does not set the %result variable. e.g: `$(%isdir:WIN1)` = 1 (assuming that you have a hard disk, otherwise it returns 0)
%isfile	Returns 1 if the operand represents a valid file (but NOT a directory). This is similar to, but not the same as, the FTEST command: The %result variable is not set, and the test will fail if the name of a directory is supplied as an operand. e.g: `$(%isfile:WIN1_Boot)` = 1 (or 0, if not there)
%flen	File length in bytes. e.g: `$(%flen:sh_c)` = 15314

%fupdt	File update date in long format. e.g: `$(%fupdt:sh_c)` = 1998 Mar 21 01:23:23
%fbkdt	File backup date in long format.
%fvers	File version number.
%fdat	File dataspace.
%ftyp	File type.
%daten	Not really a file function, this converts a long date format, of the type returned by either fupdt or fbkdt into a serial number of the form YYYYMMDDhhmmss to enable the comparison of date values. e.g: `$(%daten:%fupdt:sh_c)` = 19980321012323 The daten function can also convert the value returned by the %longdate Internal Variable into a string of this format.
	Math Functions
	Syntax: `$(%func params:operand)` Math functions perform **very** simple aithmetic operations on their operand. The math is performed in Long Integer mode.
%add	Add to the operand. e.g: `$(%add 2:2)` = 4
%sub	Subtract from the operand. e.g: `$(%sub 3:2)` = 1
%mul	Multiply the operand by the parameter. e.g: `$(%mul 2:8)` = 16
%div	Divide the operand by the parameter. e.g: `$(%div 2:7)` = 3
	Other Functions
%c68	Returns 1 if the last program executed at the command line was identified as being compiled with C68.
%file	Returns the full name of the last file to be executed at the command line. Useful for checking whether the shell is finding and running the right file.

Note: If you begin any command line with the up-arrow character (^), this will inhibit the Shell's substitution processing, and all strings will passed to the commands exactly as typed. This might be useful for some external commands that need to use the symbols recognised by the Shell.

Special Input Symbols

For maximum flexibility, and to remain in keeping with the UNIX philosophy of programs that allow piping through their Standard Input and Standard Output channels, the Shell uses very simple character input. Hence, there are no special function keys, and the editing is very basic. However, there are some special characters that help to make life easier:

Character	Meaning
=	A single equals sign typed as a command, followed by [ENTER] repeats the last command line.
/	A single slash typed as a command, followed by [ENTER], calls up the last command for editing using the standard QDOS/SMS Line Editing Trap - so, if you use SMSQ, you will have access to a reasonably good line editor.
#	Any lines beginning with a hash are treated as a comment and are completely ignored by the Shell, except for one exception: A comment of the form **#requires n.nn** (there can be no surplus spaces anywhere in the line) will check the Shell version number and exit if it less than that specified by n.nn; this is useful for writing batch files that are dependent upon features present in a specific version of the Shell (but note that versions prior to 1.12 will ignore the line completely).

External commands

These are commands that cause one or more jobs to be loaded and run. The simplest form of external command consists of the name of the program followed by its parameters e.g:

```
    make sh
```

which is equivalent to the TK II command ...

```
    EW 'make';'sh'
```

However, the Shell performs a more rigorous search to identify and load the file. It tries a number of alternatives before finally giving up, including appending default extensions to the name that you have supplied. This is the sequence that the Shell follows when trying to load an external command. Note that The Shell recognises an extension separated from the main filename either by an underscore (_bat) or a period (.bat). The period is searched for first.

1. Before doing anything else, the Shell searches for an Executable Thing that has the name specified. If one is found, load it. Otherwise it moves onto the next step ...
2. The Shell tries to open the file, in this order: (a) Name, (b) Program Directory + Name, (c) Data Directory + Name
3. If it does not succeed, the Shell then repeats this sequence with the extensions _bat, _sh, _obj, _exe, _pl and finally _rex appended to the name.
4. If the file that was found has a _bat or _sh extension, the Shell will load another copy of itself and redirect the file to the Standard Input of that copy. The Shell checks for the existence of an environment variable called SHELL to determine what program it should load to process a batch file. If this variable is not set, the Shell will try and load a file called Sh from the Program Directory.
5. If the file found has no extension, or an extension of _exe or _obj, and the file has a valid executable type header, it will be loaded as described later in this document.
6. If the file has the extension _rex, it is assumed to be a REXX program file and the Shell will attempt to load a program called Rexx, and pass the name of the file in that program's command line. The Shell will first look for an environment variable called SHELL_REX to determine the name of the REXX interpreter. If this variable has not been set, the Shell will attempt to load a program called Rexx from the Program Directory.
7. If the file has the extension _pl, it is assumed to be a PERL program file and the Shell will attempt to load a program called perl, and pass the name of the file in that program's command line. The Shell will first look for an environment variable called SHELL_PERL to determine the name of the PERL interpreter. If this variable has not been set, the Shell will attempt to load a program called perl from the Program Directory.
8. If a single filename was issued as a command to the Shell, either in a Batch File, or at the console, and this name does not match any of the internal commands, the Shell will attempt to use FileInfo II to load the application that the file is associated with. You will always have to supply the extension of the file for this work correctly, as the Shell does NOT try defaulting the extensions of data files. (Note: There is also an FX command to force The Shell to use FileInfo II).

If you prefix the name of an external program file with one of the MS-DOS (and UNIX) style directory navigation prefixes the search will be limited to the directory that results.

This happens wherever a filename is passed as the argument to an internal command, or where a filename is entered to be used as an external command or to be executed by FileInfo II, the following prefixes are recognised:

Prefixes	Meaning
\ or /	Root of the Data Directory.
.\ or ./	Force current Data Directory (Primarily used to prevent search of the Program Directory for a file).
..\ or ../	Force parent of the current Data Directory.

Examples:

Data Directory	Filename	Result
WIN1_DOCS_	\Boot	WIN1_Boot
WIN2_TX_RUN_	../tx_o	WIN1_TX_tx_o
FLP1_NOT_READY	./YET	FLP1_NOT_READY_YET

The slash "/" and backslash "\" characters may be used interchangeably and will be interpreted as having the same meaning. This is rather unfortunate for those people who use these characters in their filenames - however they are only recognised at the START of a name.

Note that the directory navigation prefixes will probably NOT be recognised by external commands (although LS v1.01 does recognise them it uses the default C68 command line processing, so to pass a backslash you have to use two of them - this was the main reason why I added the support for slash as well).

All of this means that the Shell may take a while finding the right file - especially if you use Phil Borman's superb PATH driver. On the other hand, it is an extremely flexible search method, and it keeps the total amount of typing down to a minimum.

When the program file has been identified and loaded, the Standard Input, Error and Output channels for the Shell are passed to the executed job (and can be re-directed with < and > provided the executed program handles the redirection). The executed job is 'owned' by the Shell. If the job completes with a non-zero return code a line of the form ...

Return code: nnn

... is written to the Shell's standard output, where 'nnn' is the return code as a signed decimal number.

The equivalent of

```
EX 'make';'sh'
```

is:

```
make sh &
```

In this case no Standard Input/Output channels are passed to the executed Job and the executed Job is 'owned' by Job 0 - i.e. is completely independant of the Shell. As soon as the Job is started a line of the form ...

Job nnn:mmm

is written to the Shell's Standard Output, where 'nnn' is the Job number and 'mmm' the Job tag.

Chaining external commands.

Several commands can be supplied on a single line:

```
    make sh ; make echo
```

is equivalent to the two lines ...

```
    EW 'make';'sh' EW 'make';'echo'
```

similarly ...

```
    make sh & make echo
```

is equivalent to:

```
    EX 'make';'sh' EW 'make';'echo'
```

and ...

```
    make sh & make echo &
```

is equivalent to ...

```
    EX 'make';'sh' EX 'make';'echo'
```

Piping the output of one job to the input of the next is done in MS-DOS and UNIX fashion with ...

```
    make sh | echo sh_res
```

This causes 'make' and 'echo' to be executed with 'make' owned by 'echo' and 'echo' owned by the Shell. The Standard Input of the Shell is passed to 'make', the Standard Output of make is piped to the Standard Input of 'echo' and the Standard Output of the Shell is passed to 'echo'.

this is roughly equivalent to TK II ...

```
    EX 'make';'sh' TO 'echo';'sh_res'
```

The pipeline may involve more than two jobs: e.g.:

```
    make sh | echo sh_res | echo ser1h
```

Each Standard Output is piped to the Standard Input of the Job on its right and is owned by the job on its right; with the Shell owning the last Job in the chain. The Shell's Standard Input is passed to the leftmost Job and Standard Output to the rightmost Job.

To pipe Standard Output AND Standard Error use a command of the form ...

```
    make sh |& echo sh_res
```

This does not have a TK II equivalent.

'|' and '&' may be combined ...

```
    make sh | echo sh_res &
```

Will execute 'make' and 'echo as background Jobs with 'make' owned by 'echo' and 'echo' owned by Job 0. In this instance 'make' is given no Standard Input and 'echo' no Standard Output.

A more complex combination of '&' and '|' is ...

```
    make sh | echo sh_res & make echo | echo echo_res
```

This is treated the same as the two command lines ...

```
    make sh | echo sh_res & make echo | echo echo_res
```

Internal commands

These are commands that are handled by the Shell itself. They always appear one to a line. If internal commands are combined with each other or with other external commands they are treated as external commands of the same name. Also any internal command prefixed with a '!' is treated as an external command of the same name (without the '!').

There is one special variation of internal command known as an alias. Aliases are used to map short names to more complex commands (which can be internal or external). See the ALIAS command for details.

List of Internal Commands

- ALIAS
- CALL
- CD
- CDD
- CPD
- DO
- DTEST
- ECHO
- EXIT
- FOR
- FTEST
- FX
- IF
- KILL
- LET/SET
- MD
- OPT
- PAUSE
- PRINT
- PROMPT
- VER
- WAIT
- Screen Commands

EXIT

Exits from the Shell. QUIT is accepted as a synonym.

Syntax: `EXIT {n} | QUIT {n}`

`EXIT` alone exits the Shell with a zero return code, while `EXIT n` exits the Shell with a return code of n (a signed decimal integer).

CALL

Execute the commands in a batch file, remaining in the current Shell session.

Syntax: `CALL Filename {Params}`

When a batch file is executed by typing its name, a secondary instance of The Shell is created to run it. This may not always be the behaviour that you wanted. In particular, if you wish to have the batch file make changes to your local settings (environment variables and aliases, for example) this method will not work - because the changes are only made to the child Shell.

CALL simply opens the file, and reads lines from it until either end of file, or an EXIT or QUIT command is encountered. Each line that is read is executed as if it had been typed in at the keyboard. This is how The Shell runs its AutoExec_bat file during startup.

Any parameters that you supply after the file name will be available within the batch file as the special symbols $0 to $9 (the first ten parameters) and $- (all of the parameters concatenated together and separated by spaces).

FX

Force FileInfo II to execute a file.

Syntax: `FX FileName`

This command only does something useful if the excellent FileInfo II extension is resident. The filename will be passed to FI2 for execution. This is useful to process files that are not recognised by The Shell's command parser as being potentially executable using the FI2 method.

One of the most common uses of this might be to "execute" files whose extensions are separated by a dot rather than an underscore (perhaps copied off an MSDOS volume).

Example: `FX myarc.zip`

ALIAS

Specify a command to be executed in place of a shorthand notation.

Syntax:

To set an alias:

```
ALIAS Name=Command
```

To remove an alias:

```
ALIAS Name
```

To list all aliases:

```
ALIAS
```

The purpose of an alias is to set up a short command that is easy to remember, which The Shell translates into a longer command, perhaps with additional parameters, and then executes every time you type in the short form. This is best illustrated with an example.

Suppose that you use Jonathan Hudson's port of the MTOOLS program. To get a directory of a DOS disk using that program would require the command (for example):

```
MTOOLS -C MDIR A:
```

If you define an alias for the MDIR command, like this:

```
ALIAS MDIR=MTOOLS -C MDIR $-
```

Then you can just type:

```
MDIR A:
```

And The Shell automatically expands it to:

```
MTOOLS -C MDIR A:
```

Note: This example assumes that you have set an environment variable to tell MTOOLS where it can find its configuration file, and you have mapped A: to the relevant QDOS device, and also that MTOOLS is in the Program Directory.

All aliases are stored as environment variables with the special name SH_*, where the * is replaced by the name of your alias. The value of the environment variable contains the command string.

Any parameters that you supply to an alias are stored for access as the special symbols $0 to $9 (representing the first ten parameters). The symbol $- returns all of the parameters concatenated together (but separated by spaces).

If you want to define aliases in your Boot program, please remember that the alias name MUST be in upper case to be recognised.

It is possible to define aliases that consist of more than one command. To do this you must separate each command with the line break character "`" (back quote), but please bear in mind that any parameters that you supply to the alias, when you invoke it, will only be appended to the last command in the line. This simple example will only display a directory of FLP1_ if there is actually a disk in the drive:

```
ALIAS D1=IF $(%ISDIR:FLP1) ` LS FLP1_ ` ENDIF
```

If you want to use any of the special piping symbols in the alias, you must enclose the entire alias parameter inside double quotes.

The line `ALIAS "TYPE=MTOOLS -C MTYPE $- | MORE"`, for example, defines an alias TYPE which calls the mtools utility to display the contents of a text file on an MSDOS device. The output is piped through to the MORE filter to get a pause at the end of each screen. Without the double quotes, The Shell reports an error because it tries to execute all commands at once (this is a hangover from the original design, and may be cured in later versions).

Note: Aliases are best set up in your autoexec_bat file.

WAIT

Pause.

Syntax: `WAIT {n}`

WAIT alone waits for 5 seconds, while WAIT n waits for n seconds. n must be a positive decimal integer.

KILL

Remove a Job.

Syntax: `KILL Num Tag`

Where Num and Tag are the Number and Tag respectively of the Job that you wish to remove. This is directly equivalent to the Toolkit II command

RJOB Num, Tag.

IF .. ELSE .. ENDIF

Conditionally perform instructions according to the result of a test.

Syntax:

```
    IF Left Op Right  |  IF Value  |  IF !Value

    ..

    ELSE

    ..

    ENDIF
```

There must always be either one or three parameters to an IF command. In the former case a single value is tested for either TRUE or FALSE, while the latter performs a relational comparison between two values. In the latter case, the operator determines the type of comparison that is made. Where a single value is supplied, the test is taken as being TRUE if the value is non-zero. The NOT operator (!) may precede the value, in which case if the value is zero, the test will be TRUE. The NOT operator is only valid in this context.

The result is used to either execute commands in the IF part of the construct, or in the ELSE part. An ELSE clause is optional.

Internal variables and environment variables are compared by using the default string substitution mechanism described earlier in this document. At present spaces may NOT be included in either Left or Right. The valid operators are as follows:

Symbol	**Action**
==	Strings are equal
!=	Strings are not equal
in	The string *Left* is present somewhere in the string *Right*
begins	The string *Left* begins with the string *Right*
ends	The string *Left* ends with the string *Right*
matches	The string *Left* matches the wildcard pattern specified by the string *Right*
=	Numbers are equal
<>	Numbers are not equal
>	The number *Left* is greater than the number *Right*
<	The number *Left* is less than the number *Right*
>=	The number *Left* is greater than or equal to the number *Right*
<=	The number *Left* is less than or equal to the number *Right*

Note that Left and Right MUST represent valid numbers (integer or floating point) in order for the tests relating to numbers to work correctly.

Examples:

```
    FTEST Flag_txt

    IF $(%result) = 1

    PRINT Flag file found

    ELSE

    COPY CON Flag_txt

    ENDIF
```

Or, you could do this ...

```
    IF $(%isfile:Flag_txt)

    PRINT Flag file found
```

etc.

Note that, in this example, COPY refers to an external command that performs this function, COPY is not a command that is intrinsic to the Shell.

You cannot nest IF tests. The Shell is not a programming language, nor does it seek to duplicate or replace the excellent S*BASIC or REXX languages which are far more suitable for complex Job control activities.

ECHO

Enable/disable the echoing of commands when running batch files.

Syntax: `ECHO ON | OFF`

By default, echo is disabled when running batch files. This prevents each command from being displayed on Standard Output before it is executed, and is usually neater. However, for debugging purposes, it can be useful to show the commands.

PRINT

Display a string on stdout.

Syntax: `PRINT AnyText`

This command allows you to send messages to the Shell's Standard Output. It is of most use for reporting progress in batch files. Because the substitution of special symbols takes place before a command is executed, the values of environment variables may be displayed very easily ...

```
    PRINT $(SHELL)
```

VER

Display the Shell version number.

Syntax: `VER`

Displays version number of the Shell on Standard Output. The version number can also be accessed with the %ver internal variable, and as %v in the prompt string. You can also make the execution of a batch file dependent upon a specific shell version with the special comment `#requires`.

PROMPT

Set the prompt characters.

Syntax: `PROMPT String`

By default, the Shell displays a prompt of ">" whenever it is expecting input from the console. (Although this default can be changed by setting the environment variable SHELL_PROMPT to a string that conforms to the same format set out below).

Any string can be used here, and spaces will be significant. You can also use the special symbols %d and %p - which will be replaced by the current Data Directory and the current Program Directory, respectively. Use %% to get a single percent sign into the prompt. Other symbols are listed in the table below.

Special symbols:

Symbol	Meaning
%d	Current Data Directory
%p	Current Program Directory
%D	Current date in the form: Day Month Date Year
%T	Current time in the form: HH:MM:SS
%v	Current Shell version number
%[Name]	Current value of the named environment variable
%[%Name]	Current value of the named Shell Internal

Note that %[Name] and %[%Name] are there because any attempt to use $(Name) and $(%Name) as part of the argument to the PROMPT command would cause these values to be substituted immediately, rather than at the time the prompt is actually printed.

Important: Case is significant - %d is not the same as %D.

LET/SET

Set/Remove a local environment variable.

Syntax:

`LET Var=Value` (or `SET Var=Value`)

Assign Value to Var.

`LET Var=` (or `SET Var=`)

Assign an empty string to Var.

```
    LET Var (or SET Var)
```

Delete Var from the environment.

This command is used to assign values to the Shell's environment variables. Because of the way in which C68 programs use their own UNIX-like private environment, the LET command does not affect the master environment as seen by S*BASIC (use the Set external command to do this). However, by default the environment will be passed down to any C68 programs that the Shell executes (see the OPT command for more details). The external command LENV can be used to display the contents of the environment.

Note: As can be seen above, Shell recognises either LET or SET for this command; they do the same thing!

FOR

Execute a command for every file that matches.

Syntax: `FOR WildCard {,WildCard} Command`

For every file whose name matches the wildcard specification supplied, the remainder of the line is executed as if it were an ordinary command being submitted to the Shell. The special symbol $*, if used in the command, will be translated to the name of the current file.

You can specify multiple wildcard specifications in a single command by separating each one with a comma. In this case, the command will process each specification in turn.

It is possible to specify multiple commands by entering them all on the same line and using the special line break character "`" (back quote). Note that the line break character is valid only for the FOR, DO and ALIAS commands.

Example:

```
    FOR *_txt FOLD $*_new
```

For every _txt file in the current directory, call the external FOLD command with that file as stdin, and stdout sent to a file with the same name, but _new appended.

There is also another special variable whose value only has meaning when used in a FOR command. The symbol $^ returns the current filename minus the extension (i.e. all characters starting from the rightmost underscore are stripped off). The following example shows how this facility can be used to change file extensions:

```
    FOR *_txt FOLD $^_new
```

The difference between this example, and the first is that the first method would create a file called Foo_txt_new, whereas the second produces Foo_new.

The Shell does not perform any substitution of special symbols ($A etc.) on a FOR command, until the line is actually executed - so the substitution occurs once for each matching file.

Note that the string functions listed in the section on Special Symbols are most useful here. For example ...

```
    FOR *_h PRINT $(%part 4:$*)
```

... If, say, you were in a directory WIN2_C_SH_, this would display a list of just the names of each _h file - directory, and no extension.

And finally, you can specify multiple sets of files with a single command ...

```
    FOR *_c,*_h cp *$ FLP1_
```

... copies all C source files to FLP1.

```
    FOR *_c IF !$(%isfile:\$_txt) ` PRINT $/ ` ENDIF
```

This unlikely looking command lists all of the _c files that do not have a corresponding _txt file. Note the use of the line break character (back quote) to break up the three lines of the IF test.

See the note on quoted strings under the ALIAS command for the method that should be employed to include job piping symbols into the command(s) to be executed.

DO

Execute a command whilst iterating a loop variable.

Syntax: `DO Value | From:To {,Value | From:To ...} Command`

This very simple looping construct will execute the supplied command for every value in the From and To range. If both From and To are numbers, an integer sequence is processed. If either From or To are characters, the loop is conducted for a single character that takes on each value in the range.

There can be no spaces between the From and To - which means that a space cannot be used as part of a character loop. Also, the comma is used to separate multiple DO ranges or values.

If From is less than To, the loop counts down instead of up.

The current value of the loop counter is accessed by the reserved symbol "$+".

It is possible to specify multiple commands by entering them all on the same line and using the special line break character "`" (back quote). Note that the line break character is valid only for the FOR, DO and ALIAS commands.

Examples:

```
DO 1:10 PRINT $+
```

Print the numbers from 1 to 10.

```
DO a:f RM file$+_txt
```

If you have the external RM command, this will delete the files filea_txt, fileb_txt .. filef_txt.

Multiple ranges or values can also be used:

```
DO 1,3,5,7

DO a,s,d,f

DO 1,6:12,15,20:25
```

Note that you cannot have any spaces anywhere in the list of DO ranges as this will be interpreted as the start of the command that you wish to DO.

See the note on quoted strings under the ALIAS command for the method that should be employed to include job piping symbols into the command(s) to be executed.

OPT

Set Shell options.

Syntax: `OPT {Option}`

This command is provided for easy access to the special environment variable SHELL_OPTS. This variable holds a string of switches which normally take the form +X - where X is an upper or lower case character. Case is significant.

Note that you can preset the Shell Options by setting the value of SHELL_OPTS before the Shell is started.

OPT followed by any string checks the SHELL_OPTS variable to see if that string is already present. If so, it removes it, otherwise it appends it to the end of the variable.

Examples:

Before	Command	After
	OPT +d	+d
+d	OPT +d	
+d	OPT +e	+d+e

Passing no options will display the current option string. This can also be done by PRINTing $(SHELL_OPTS).

The following options are recognised:

Option	Result
+d	Force automatic update of directories. (Whenever you change a directory in the Shell, it is also changed in QDOS/SMSQ). See the DIR command.
+e	Force the environment to be passed to an external command.
-e	Prevent the environment from being passed to an external command.
+u	Force directory names for CD and MD into upper case.
+l	Force directory names for CD and MD into lower case.

The +e and -e switches deserve some explanation. C68 programs support a mechanism for passing the environment between parent and child.

Prior to v2.01 of C68 the only transfer supported was for the three directory strings. The Shell tries to determine the nature of any program file that it loads as an external command. If it thinks it has been created with a 'new' version of C68, it will pass a long pointer to the environment on the job's stack - otherwise it will pass the three directories. Setting +e forces the environment to be passed regardless of the program file, whereas -e prevents the Shell from ever passing the environment. Note that the Shell uses one method or the other: environment OR directories. Not both and not either. The -e option, if present, will override +e if that is present as well.

+e might be useful if the Shell fails to recognise a 'new' C68 program, while -e is useful to stop a child process from inheriting the Shell's environment. Note that the command "OPT -e" does not remove a +e setting, as this is not how the OPT command operates. To remove +e issue another OPT +e.

CD

Change the Data Directory.

Syntax: `CD Directory`

Changes the current Data Directory to that specified. If the directory does NOT begin with a valid device name, the new directory is appended to the current one, otherwise it replaces it in its entirety.

Thus, if the current Data Directory was WIN1_DOCS_, and you issued the command CD TECH, the new directory would become WIN1_DOCS_TECH_, but issuing the command CD RAM1_Scratch would change it to RAM1_Scratch_.

CD also recognises some (primarily) MS-DOS and UNIX style special characters. The backslash symbol (\) means go to the root of the device that is currently the Data Default, while double dot (..) is translated into "move to the parent directory". Thus ...

Current	Command	New Value
WIN1_HI_THERE_	CD ..	WIN1_HI_
FLP1_BAD_SECTORS_	CD \	FLP1_
RAM1_PROCS_	CD \TXT	RAM1_TXT_
WIN1_DOCS_TECH_	CD ..\FICTION	WIN1_DOCS_FICTION_

For convenience, you may omit the space between CD and its parameter, IF the parameter begins with \ or .. - this is compatible with the Command Interpreter in MS-DOS.

The Shell maintains its own "local" directories which it inherits from S*BASIC when it is initially started. Consequently any changes to the directories within the Shell are not reflected back in S*BASIC. Programs compiled using C68 have a mechanism for passing directories from a parent to a child - although this method changed from version 2.01 onwards to enable a pointer to the environment variables to be passed instead.

The Shell supports some additional syntax that will force the S*BASIC directory to match the setting in the Shell, and this is done by prefixing the directory name with an '@' sign. e.g:

```
    CD @TEXT
```

... will act as documented in the table above, but will also set the S*BASIC Data Directory to the new string. For convenience, you may omit the space between the CD and the '@'. CD@ alone, will copy the Shell's Data Directory setting back to S*BASIC.

In addition, all three directory change commands will accept a single "=" sign as a parameter, this causes the Shell to set its local directory equal to whatever is currently set by QDOS/SMSQ. This can be quite useful for when you want to "exploring" away from the startup directory, because a single CD= takes you right back.

If you like your directory names to be in upper or lower case, you can force this automatically by setting the Shell Options +u and +l (see the OPT command for details).

Another Shell Option that affects this command is +d (also documented under the OPT command). This facility lets you force the automatic synchronisation of the directories, and is very useful if you are using primarily non-C68 external commands.

CPD

Change the Program Directory.

Syntax: `CPD Directory`

This command behaves exactly like CD, except that it operates on the Program Directory.

CDD

Change the Destination Directory.

Syntax: `CDD Directory`

Although this command behaves in exactly the same way as the CD and CPD commands, it is unlikely that you will ever need to use it in this way.

Normally CDD is used to set an output device for spooling and printing operations. However, it can specify a directory for use in copy operations.

MD

Make a new directory.

Syntax: `MD Directory`

This command is used to create a new directory on a Level 2 device. As with the CD command, MD also recognises some special symbols that enable you to specify where the directory will be created.

If the directory does not begin with a valid device, the directory is created in the current Data Default, otherwise it is created on the device and location that you specify. You can use the \ and .. symbols to force creation of a directory off the root, or at the same level (i.e. below the parent of the current directory). Here are some examples:

Command	Action
`MD Hello`	Create Hello below Data Default
`MD RAM1_ME`	Create directory ME on RAM1_
`MD \NEW`	If the Data Default was currently WIN1_WORK_ this command creates a directory WIN1_NEW_
`MD ..\Asm`	If the Data Default was currently WIN1_WORK_BAS_ this command creates a directory WIN1_WORK_Asm_ (but does not change the Data Default)

As with CD, you can omit the space between the command and its parameter, if that parameter begins with \ or ..

You can force new directory names to upper or lower case by using the Shell Options +u and +l respectively (see the OPT command).

PAUSE

Pause the execution of a batch file and wait for the user to hit a key.

Syntax: `PAUSE {Message}`

This command can be useful to pause at the end of a series of commands to indicate that the operation has been completed. Optionally, a message can be printed. If the Shell has no CONsole or SCReen channels open for Standard Input or Standard Output, it will open a CON channel specifcally for this purpose, and close it afterwards.

The ASCII code of the key that the user pressed is stored in the Shell internal variable %lastkey - this enables you to prompt for a user action and perform commands conditionally depending upon the key they hit.

FTEST

Test for the existence of a file or directory.

Syntax `FTEST File | Directory`

Checks whether the specified file or directory actually exists, and sets the value of the internal variable %result accordingly (1= TRUE, 0=FALSE). The %result variable may then be tested by a subsequent IF test.

If you specifically wish to test just existence of a directory, you should use the DTEST command instead, as FTEST will not tell you whether, say, WIN2_C_SH is a file called C_SH on WIN2, or a directory SH in a directory C on WIN2!

There is also a corresponding file function (%isfile) which has similar functionality, and which you can use to just test for the existence of a file.

DTEST

Test for the existence of a directory.

Syntax `DTEST Directory`

Unlike FTEST, this command checks whether the argument supplied is an existing directory (but NOT a file). It can also be used to test whether a device exists, and is a directory device. It sets the value of the internal variable %result accordingly (1= TRUE, 0=FALSE). The %result variable may then be tested by a subsequent IF test.

There is also a corresponding file function (%isdir) which has the same function.

Examples:

```
    DTEST WIN2_C_SH

    DTEST FLP1
```

Important: Note that no trailing underscore should be used.

Screen-Related Commands

The internal commands in this section are only valid if the Shell's Standard Output is a SCR or CON device. If the Standard Output is not SCR or CON these commands are treated as external commands of the same name.

List of screen-related commands

- BORDER
- CLS
- CON
- INK
- PAPER
- WINPOS
- WINSIZE
- WMOV
- Other Internal Commands

CLS

Clear the Standard Output window.

Syntax: `CLS`

INK

Set the ink colour of the Standard Output window.

Syntax: `INK Colour`

Directly equivalent to the S*BASIC command of the same name. The new ink setting takes effect from the next characters to be displayed.

PAPER

Set both the paper and strip colours of the Standard Output Window.

Syntax: `PAPER Colour`

Directly equivalent to issuing the S*BASIC PAPER and STRIP commands. The new paper setting takes effect from the next characters to be displayed.

BORDER

Set the border colour and width of the Standard Output window.

Syntax: `BORDER Width Colour`

This command can be used to set the width (in pixels) and colour of the Shell's main CONsole window. In fact it drops right through to the CON command, which is described below, to do all the work. You can just change the colour of the border by passing -1 for the width. Or, you can just change the width by leaving off the colour parameter.

WINPOS

Set the position of the Standard Output window.

Syntax: `WINPOS XPos Ypos`

This command positions the top left corner of the Standard Output window at the indicated Xpos and Ypos pixel coordinates, leaving the width and height unchanged. To just change the X position, omit Y. To just change Y, pass -1 for X. This command drops through to CON (documented below) to do the work.

WINSIZE

Set the size of the Standard Output window.

Syntax: `WINSIZE Width Height`

Redefines the width and height of the Standard Output window, the position of the top left corner is left unchanged. To just change the Width, omit the height. To just change the Height, pass -1 for the Width. This command drops through to CON (see below) to do the work.

WMOV

Syntax: `WMOV`

This command allows you to interactively alter the window size and position. After entering the command you will be advised which keys to use to move the window, either by pixel or character increments, or to re-size it. Once the window is positioned and sized the way that you want it, pressing [ENTER] will accept the changes or use [ESC] to revert to the previous settings.

CON

Set all CONsole parameters in one go.

Syntax: `CON Width Height Xpos YPos BWidth BColour Paper Ink XShad YShad`

Use this command to set size, position, colour etc. at the same time. Any trailing parameter may be omitted, and the defaults will be used. To leave a particular parameter unchanged, pass the value -1.

CON on its own will reset all of the window defaults to their startup value. Quite that this might do, if the Shell was passed another job's console window at startup is likely to be the source of some amusement.

The command `CON -m` is a synonym for WMOV; it will enter interactive mode, allowing you to move and size the window using the cursor keys.

Note that, under the Pointer Environment, this command also resets the window's outline.

Some Useful External Programs

To do anything really useful with the Shell you need some external programs. There are some very good programs supplied with the C68 compiler (for instance, ECHO which copies its command line to Standard Output - effectively functioning as a PRINT command for the Shell), and the GNU Text Utilities also work well with it. I also use a collection of very useful programs written by Richard Kettlewell, which come bundled with the EMACS disk from QUBBESoft P/D - these include COPY, DIR and REN. All of the latest versions of my QL Toolbox 98 range of Pointer Environment utilities recognise when they've been passed three channels and a command-line and work just as if they were ordinary commands. And, of course, the programs that make up the very good C68 suite can all be executed directly from the Shell.

So that you can use this program straight away, I recommend downloading Richard's utilities and putting them in the Program Directory, where they will then function just as if they were ordinary commands.

I have also written some external utilities specifically to be used from the Shell. These are available as separate archives, complete with source code - so you can fiddle around with them to your heart's content.

LENV

LENV is a very simple program to display the contents of the local environment (just like the S*BASIC ENV_LIST command, in fact). Again, this file should be placed in your program directory. A companion utility to this is SET, which operates on the master environment.

LS

LS gets its name from the UNIX command to display a list of files. It is very similar to Richard Kettlewell's DIR, except that it displays additional information such as the file type and dataspace (if the file is executable). LS also strips the directory off the filenames before displaying them, making its output a lot easier on the eyes. To find out what parameters LS will accept simply execute it with a -? parameter.

Batch Files

A batch file is a simple plain text file that contains a list of commands which the Shell will try and execute. Internal and external commands can be freely mixed, and IF tests allow very basic decision making. When you execute a batch file, either directly by, say, `EX 'sh';'<TrashMap_bat'`, or by simply typing its name in the Shell, any extra parameters can be accessed as $a to $z.

Example:

```
EX 'sh';'<TrashMap_bat WIN1_ WIN1_ /all'
```

$a=WIN1_

$b=/all

```
DirClone WIN1_ RAM1_
```

$a=WIN1_

$b=RAM1_

You can pass up to 26 parameters in this way. (Which should be more than enough for most people. However, if you want more, just turn to the source code ...)

Here is an example, a file called DirClone_bat which uses my DirList and MakeDirs utilities to clone a directory tree from one device to another ...

```
    print About to clone $a to $b

    pause Hit [ENTER] to start ...

    if $(%lastkey) = 10

    dirlist $a /x /b | makedirs nul $b /x /b

    endif
```

Next Steps

If you want to do more complex scripting you will find that QL REXX is an excellent vehicle for this, and it integrates with the Shell very well. The Shell knows about _rex extensions and will automatically try and pass control to REXX if it is asked to load such a file. You can get a copy of REXX from several of the BBSs. The Shell passes its channels to REXX, so the whole thing works very smoothly.

If you're running SMSQ you will be pleasantly surprised by what happens when you type the command SBASIC. All of a sudden you're in an SBASIC interpreter with the same window that the Shell had been using.

Also, because QL Shell tries to find Executable Things first, every QPAC 2 menu can be invoked just by typing its name!

With FileInfo II loaded as well, not only can you associate _bat files with the Shell, but you can load any file into its application just by typing its name!

Acknowledgments

- P.J. Taylor, the original author of the Shell.
- David Gilham who updated the Shell to v1.11 with WMOV and some bug fixes.
- Dave Walker who maintains the C68 Compiler suite.

CHANGE HISTORY

Details of the changes made to the Shell prior to version 1.12 can be found in the file whatsnew_txt which is distributed with the program.

v1.12

- Fixed problem with matching nested pairs of brackets that incorrectly parsed expressions like $(%isfile:$(filename)).
- "set" is now a synonym of "let".
- If an autoexec_bat startup command file is not found, the Shell will now look for a file called autoshell_bat.
- A special comment of the form #requires n.nn in a batch file will stop execution of the file if the Shell version number is less than that specified.

Table of Contents