Script Reference

Microsoft, Windows, MS are registered trademarks of Microsoft Corp.
IBM and PC are registered trademarks of IBM Corp.

Glink Enterprise Edition, version 8.6
Glink Professional Edition, version 8.6
© Gallagher & Robertson A/S 1987-2019
All Rights Reserved

GALLAGHER & ROBERTSON A/S, Grini Næringspark 3, N-1361 Oslo, Norway
Tel: +47 23357800
www: http://www.glink.com/
e-mail: support@glink.com

Contents

Contents i

General 1
Overview 1
Starting a script 2
Script directories 5
Script file format 6
Nesting of scripts 7
Compiled scripts 7
Termination script 8

Script variables 11
Normal variables 11
Numeric variables 12
Built-in variables 13
The $STATUS variable 19
Pattern variables 19
File variables 20

Script command categories 21
Alphanumeric handling 21
Compiler and debugging commands 21
Configuration control 22
Control structures 23
Dial directory 23
File I/O commands 24
File transfer and control 24
Host interaction commands 25
Key definitions and handling 26
Menu handling 26
Screen and cursor control 27
System-related commands 27
Timing commands 28
User input commands 28
Variable handling commands 28

Windows-specific commands 29
Commands for backwards compatibility 30

Script commands 31
The #ELSE directive 31
The #ENDIF directive 31
The #IFDEF directive 32
The #IFNDEF directive 33
The ABORT command 33
The ACTIVATE command 33
The ADD command 34
The ADDMENU command 34
The ADMSHELL command 35
The APPEND command 35
The ASSIGN command 35
The BEEP command 36
The BEGIN command 36
The BINARY command 36
The BREAK command 37
The BUILDMENU command 37
The BUTTON command 38
The CALC command 40
The CALL command 41
The CAPTURE command 42
The CASE command 42
The CD command 43
The CFIX command 44
The CFXW command 44
The CHAIN command 44
The CHANNEL command 45
The CLEAR command 45
The CMPNUM command 45
The COMPARE command 46
The CONCAT command 46
The CONFIG command 47
The CONNECT (modem) command 47
The CONTEXT command 48
The CONVERSE command 49
The COPY command 49
The CRDB command 49
The CRDW command 50
The CREAD command 50
The CRLF command 50

The CSWITCH command 51
The CTYPE command 51
The CURSOR command 52
The CXRESTORE command 52
The CXSAVE command 53
The DBOX command 53
The DCHANGE command 54
The DDEADVISE command 55
The DDECLOSE command 55
The DDEEXECUTE command 55
The DDENAME command 56
The DDEOPEN command 56
The DDEPOKE command 56
The DDEREQUEST command 57
The DEBUG command 57
The DEFAULT command 58
The DEFINE command 58
The DELAY command 58
The DELMENU command 59
The DFIND command 59
The DIAL command 60
The DISCONNECT (modem) command 60
The DIVIDE command 61
The DMARK command 61
The DOMENU command 62
The DOS command 62
The DOSN command 63
The DOWNLOAD command 64
The DPATTERNS command 64
The DREAD command 64
The DSCREEN command 65
The DTENTHS command 65
The DTIME command 65
The DUNMARK command 66
The DVARIABLES command 66
The DWHENS command 66
The ECHO command 67
The EIGHTBIT command 67
The ELSE command 67
The EMULATE command 68
The ENABLE command 68
The ENDBUILD command 68
The ENDIF command 69

The ENDSWITCH command ... 69
The ENDWHILE command ... 69
The ERASE command ... 69
The ERRORGOTO command ... 70
The EXECUTE command ... 70
The EXISTS command ... 70
The EXITSWITCH command ... 71
The EXTRACT command ... 71
The FCLOSE command ... 71
The FCODE command ... 72
The FILTER command ... 72
The FIND command ... 73
The FIX command ... 74
The FLOC command ... 74
The FLUSH command ... 74
The FNDEXEC command ... 75
The FNEXT command ... 75
The FOPEN command ... 75
The FPOS command ... 77
The FRDBLOCK command ... 77
The FRDCHAR command ... 77
The FRDLINE command ... 77
The FSEARCH command ... 78
The FSEEK command ... 78
The FSIZE command ... 79
The FSKIP command ... 79
The FTP command ... 79
The FVERSION command ... 80
The FWTBLOCK command ... 81
The FWTLINE command ... 81
The GETDATE command ... 81
The GETENV command ... 82
The GETFILE command ... 82
The GETKEY command ... 83
The GETLENGTH command ... 84
The GETMACRO command ... 84
The GETTIME command ... 84
The GETVALUE command ... 86
The GETWORD command ... 86
The GOSUB command ... 87
The GOTO command ... 88
The GPARAM command ... 88
The GPROFILE command ... 89

The GWCONNECT command 89
The HALT command 91
The HOST command 91
The ICON command 91
The IDLE command 92
The IF command 92
The INCLUDE command 96
The INFILE command 96
The INPC command 97
The INPUT command 97
The INVISIBLE command 98
The ISOCONNECT command 99
The ISSERVICE command 99
The KEYBOARD command 100
The KEYKERMIT command 100
The KEYS command 100
The LABEL command 103
The LAYOUT command 103
The LCASE command 103
The LINE command 104
The LOCAL command 104
The LOG command 104
The MANDIAL command 104
The MARK command 105
The MATCH command 106
The MBAR command 106
The MCURSOR command 107
The MD command 108
The MDIAL command 108
The MENU command 109
The MESSAGE command 110
The MFONT command 110
The MINIT command 111
The MODE command 111
The MOK command 112
The MONO command 112
The MOP command 112
The MOPC command 113
The MOPTION command 113
The MOVEWINDOW command 114
The MPOS command 114
The MSGBOX command 115
The MTEXT command 115

The MULTIPLY command 116
The MVSCROLL command 116
The NAME command 117
The NETCONNECT command 117
The NETDISCONNECT command 118
The NEW command 118
The NOMENU command 118
The OBJECT command 119
The OEM command 119
The OLE command 120
The ON command 122
The ONLINE command 125
The PACE command 126
The PARAM command 126
The PARITY command 126
The PATTERN command 127
The PAUSE command 127
The PERFORM command 128
The PICK command 130
The PLAY command 130
The POPUP command 131
The PORT command 131
The POS command 132
The PPROFILE command 132
The PREMOTE command 133
The PRINT command 133
The PSET command 133
The PUTFILE command 134
The QUIT command 134
The RATR command 135
The RCVLINE command 136
The RCVTURN command 137
The RD command 137
The RDIAL command 137
The RECEIVE command 138
The RECS command 139
The REMENU command 140
The REN command 140
The REPLACE command 140
The RESET command 141
The RETCALL command 141
The RETURN command 142
The RFORM command 142

The RKEY command 142
The ROLL command 143
The RSBK command 143
The RSCR command 144
The SCAN command 145
The SCREEN command 145
The SECURE command 146
The SEND command 146
The SEPMENU command 147
The SERVER command 147
The SET command 147
The SETMACRO command 162
The SHELL command 162
The SHOW command 163
The SNDLINE command 163
The SPEED command 163
The SPLIT command 164
The STITLE command 164
The STRACE command 164
The STRIP command 165
The SUBRIGHT command 165
The SUBSTR command 165
The SUBTRACT command 166
The SWITCH command 166
The TCKEY command 167
The TIMEOUT command 167
The TITLE command 168
The TRACE command 168
The TRANSMIT command 169
The TRIM command 169
The TRNLINE command 169
The TRUNCATE command 169
The TSMDIR command 170
The UCASE command 170
The UNMENU command 171
The UPLOAD command 171
The URLSHOW command 171
The WELCOME command 172
The WHEN command 172
The WHILE command 173
The WINDOW command 174
The WKEY command 175

The DBOX command 177
General 178
Dialog units 180
Dialog box controls 181
Dialog box elements 188
Automatic group boxes 188
Bitmap buttons 189
Bitmaps 190
Check boxes 191
Combo boxes 192
Centered text 193
Default pushbuttons 194
Edit text 195
End of group marker 196
End of horizontal group 196
End of vertical group 197
Group boxes 197
Horizontal group 198
Icon buttons 199
Icons 200
List boxes 201
Left justified text 204
Pushbuttons 205
Radio buttons 206
Right justified text 207
Size buttons 208
Trackbars 209
Vertical group 210

External interface 211
Overview of extension DLL interface 211
Using external functions in a script 211
Programming external script functions 212
Data types for the DLL 216
C: character data 217
H: handle data 217
I: integer data 217
L: long integer data 217
O: script OK status 218
S: structure data 218
External values 218
Search rules 220

Examples of extension DLLs 220

Script examples 221
Simple login to bulletin board 221
More complex login 222
Login with error checking 223
'Event-driven' login 224
Menu-controlled script 225

Running VBScript or JScript files 227
Inheriting the GlinkApi object 228
Passing input parameters 229
Return values 230

Configuration file format 232

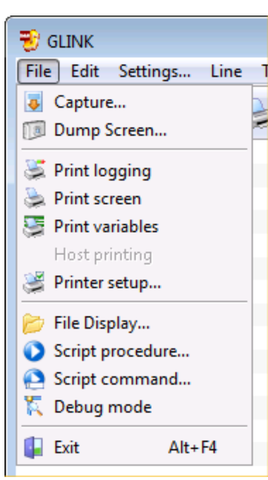
Index 259

General

Script files are a way of automatically programming your dialog with the host machine. Tasks requiring the same kind of input from your terminal each time can thus be automated. Depending upon how much work you are prepared to invest in the writing of such scripts, intelligent decisions may be made as to the correct course of action to take in a variety of situations. This can allow your PC to handle dialog with host systems in unattended mode. Also, easily programmable menus and dialog boxes allow you to provide an easy-to-use user interface to complex host systems, hiding the details of the interaction with the host from the user at the terminal. The script files used by Glink are standard PC text files that you may prepare using any normal text editor such as NOTEPAD.

Overview

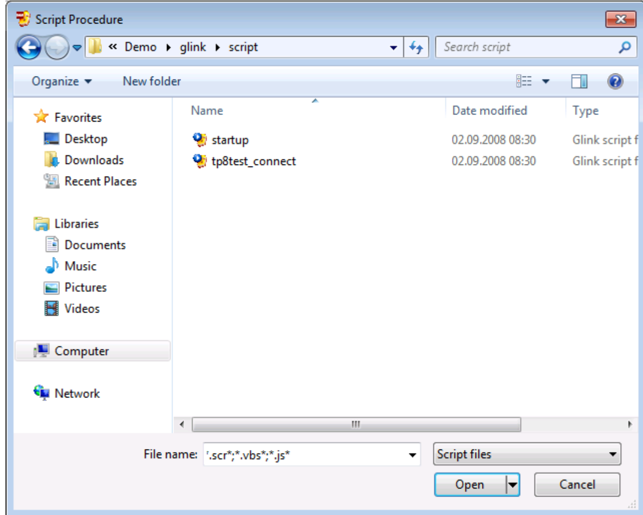
A script file is first created with your editor. It contains commands like 'send' and 'receive' to tell the emulator what to do. The script procedure is invoked using selecting the FILE/SCRIPT PROCEDURE option from the menu bar, or pressing the 'start script' key, ALT+O.



By convention, script files will have an extension of `'SCRGL'`. This allows the file window to select only the script files if you need to view your local directory to find the script. When the script has been selected, it will be 'compiled' by Glink. That is, the entire file will be read and checked for errors before the actual procedure is started, and a 'tokenized' version prepared that allows for much faster execution than would have been the case if the script had been interpreted while it was executing. If the script is OK, then Glink will start executing the commands found there, and the status line will show you the name of the script file to let you know that an automatic procedure is being performed. The script may call for Glink to wait for input from the host - any time it does this and seems to be stopped you can always get it going again by pressing any key on your keyboard. Things you type will also be sent to the host; this may be useful if you need to type something like a filename that you don't want to include in the script itself. The script can also be stopped at any time just by pressing the ESC key. Also, if the script is waiting for something to happen, for example in a receive command, you can tell it to stop waiting simply by pressing the ALT+O key.

Starting a script

There are several ways of starting a script. The simplest is to do it manually with the ALT+O command from the keyboard or from the menu bar by using the FILE/SCRIPT PROCEDURE option. The ALT+O command will display a list of the scripts in the current directory you may to execute. The script you select will then be executed, assuming it contains no errors.



Another way of starting a script is by 'linking' it to the dial directory. If a host defined in the dial directory has a script name specified as part of its configuration then this script will be started automatically each time you connect to the host in question.

Yet another way of starting a script is to configure a macro key that initializes the script using the `'^*scriptname'` syntax. If an entry there contains the `'^*'` string then the rest of the text in that line will be considered to be the name of a script that is to be executed when the appointment falls due.

The host machine may also start a script, using the defined escape sequence for that purpose (see the *Command extensions* appendix to the *User's Guide*).

A script may be run as the emulator is started; this is done using the `/S` command line parameter, either written directly in the command line, or specified using the `GLWINOPT` environment variable.

You may also start a script from the File Manager (or any other application that allows you to drag files) simply by dropping the script onto the Glink window or icon. For this to work the file name must have the extension `'.SCRGL'`, or `'.SCR'` on a short filename system. Dropping a file with any other name will cause Glink to attempt to start a script called `DRAGDROP.SCR`. If that script is found then the name of the dropped file will be provided to it in the `$FILE` internal variable.

Finally, you may also start a script from another script, using the CALL and CHAIN script commands.

A special syntax is available in any situation where a script file is to be started. If the file name starts with an equals sign (=) then the name will be considered to be a script command, rather than a file name. For example, you could type in '=DEBUG ON' in the ALT+O window to enable debugging interactively, or send '=HALT' as a script command sent from the host to terminate the emulator.

In any situation where you are starting a script you may start at a predefined location in the script file by using the form:

```
FILENAME!label_name
```

in which case execution of the script will begin at the specified label rather than at the start of the script (this syntax is ***not*** allowed if the script in question is a compiled script). You may supply parameters for the script to be executed by specifying them immediately after the name of the script (and starting label if present). These will be accessible from the script using the $PARAMETER and $Pn internal variables. For example: CALL "MYSCRIPT.SCR p1 p2".

Script directories

A script file may be in one of several directories. When a script name is specified, if it's an absolute pathname (a name with a colon or a backslash in it) then Glink will just look for that particular file. If you specify just the filename then Glink will search for the file in the following directories (in this order):

1. In the current directory.
2. In the user script directory (specified with /OU in the command line or using the GLSUSE environment variable).
3. In the script directory (specified with /O in the command line or using the GLSCR environment variable).
4. On the user directory (specified with /U in the command line or using the GLUSE environment variable).
5. On the configuration directory (specified with /CD in the command line).
6. In the Glink directory (the directory in which the Glink program resides).

Script file format

The format of the script file is simple: it consists of a list of script commands, either one per line or several on a line, separated with semicolons (;). The first word of each command is the script command. Only the first four characters of the command are checked, so if you wish you can write `'ASSI'` instead of `'ASSIGN'`, for example. Some commands (like `'QUIT'`) don't need any extra information, while others (like `'SEND'`) will need one or more extra items of information. In the case where this is a string of text, the text should be enclosed in quotes (for example, `"hello^M"`). Note that the same convention is used as for macro keys, with the caret symbol and an alphabetic character to send control characters should these be needed. You may use either single (') or double (") quotes to enclose your strings, this allows you to include quote characters inside a string, e.g. ' " '.

For script constants, the caret (^) control-character syntax is extended to support other types of format for 'awkward' constants. The complete list is as follows:

`^^`	inserts a single ^
`^X`	control character (uses the bottom five bits of the specified character only)
`^#ddd`	decimal specification
`^&ooo`	octal specification
`^$hh`	hexadecimal specification

Using the different possibilities here, this means that the ASCII CR control character may be specified in any of the following ways:

```
"^M", or "^#013", or "^&015" or "^$0D"
```

Note that the leading zero in these expressions is required; the expression must have the exact length as specified above. See the *Character equivalents* appendix to the *User's Guide* for a table of control characters.

When more than one command is specified on a line, the commands will normally be separated using semicolons as mentioned. This is not required, but is recommended, in that some of the script commands take a variable number of parameters, and use of the semicolon will then resolve any possible ambiguities.

Comments may be included in script files; this is done by using an asterisk (*) at the start of a comment. This may be done either at the start of a line or partway through the line. In any case, the rest of the line will be ignored by the script compiler. Note that this means that you MUST use quotes to enclose an asterisk that is part of a script command. For example:

```
KEYS Gr- "*" Gr- "J" * This is a comment
```

is a valid statement, showing usage of the asterisk both inside a script statement and to introduce a comment.

Scripts of any size may be compiled (subject to memory limitations) but the compiled version is limited to 64Kb of generated object code.

Nesting of scripts

A script may call another script using the `CALL` command. Other situations may arise where a script will be invoked 'inside' another script, and in general Glink will let you do this. A typical situation for this type of usage is where a script has been written as a 'monitor' for certain strings that may be received from the host and is 'idling' with the `ONLINE` command. In this case, starting a new script, either under host control or manually with ALT+O will start the new script, but when the new script is terminated, control will return to the original script. `WHEN` statements that might be active in the original script are still active in this case (assuming that the called script does not change the patterns involved, etc.).

Transfer of parameters to another `CALL`ed script may be done either using the normal variables (which are not destroyed when a script is `CALL`ed) or using `$Pn/$PARAMETER` and including the parameters in the `CALL`.

Compiled scripts

A script will normally be run directly from the text file that contains the script commands; the overhead incurred for analyzing the input is very slight for all but very large files. However, either because the script is large or for security reasons, you may wish to run 'compiled' scripts. These consist of an internal format which is more compact than the original text file and which also is encryp-ted to prevent easy analysis of the contents of the script procedure. To compile a script, simply include the command:

```
OBJECT "filename"
```

anywhere in the script, and a compiled version will be produced on the named file. The script will in this case not actually be executed. The output file, `"filename"`, may then be used in place of the original script.

When you update to a new release of the software, you will usually have to recompile any scripts that were prepared with an earlier release. If this is the case you will receive a warning if you try to use scripts requiring recompilation.

Termination script

You may supply a script to be executed whenever Glink is to be terminated by the user, simply by placing a script using the conventional name `$$TERM.SCR` in your script directory. If this script is present then it will be executed whenever a request is made to terminate Glink, and Glink will not be terminated. You will normally provide for such termination by including a `HALT` command in the script itself.

The terminate script will by default only be run the first time the user attempts to exit Glink (this is to protect you from unintentionally getting into a situation where the user is unable to terminate at all). The script may use the `SET TERM TRUE` command to request that the terminate script should be re-enabled if this is required. In the same way a script may be executed at any time to perform a `SET TERM FALSE` to disable a later execution of the terminate script.

Script variables

Normal variables

Any place that you can use a string, you can also use a 'variable'. Glink provides you with 99 variables, numbered from 1 to 99.

Values may be assigned to these directly using the ASSIGN command, or interactively using the INPUT command, up to a maximum of 255 characters per variable. Variables are referenced using the number of the variable prefixed with the percent (%) sign. You can use a variable anywhere that you can use a string, as well as in some commands that require a variable, like ASSIGN. When the command requires a variable the percent sign is optional, but we suggest you always include it for the sake of readability. For example, to send the current contents of variable 8 and a carriage return you could use:

```
SNDLINE %8
```

Also, in all situations where you are able to use a simple string, you can combine strings, variables and built-in variables (see below) by using parentheses in the following way, for example:

```
SNDLINE ($LOGIN " " $PASSWORD ";" %2)
```

Script constants, with the ^control-character syntax may also be used. For example the ASCII CR control character may be specified in any of the following ways:

```
ASSIGN %2 ("^M" "^#013" "^&015" "^$0D")
ASSIGN %3 "^M^#013^&015^$0D"
```

Variables may be specified indirectly using the contents of another variable; this is done by using the underline (_) character instead of the % character before the number of the variable. This usage requires that the specified variable contain a valid number, of course. For example, if variable %5 contains "12", then the script statement:

```
ASSIGN %11 _5
```

will assign the contents of variable %12 to variable %11 rather than using the contents of variable %5.

To enable you to make scripts more legible, a statement is provided which allows you to associate a name with any of the numbered variables. This is done with the DEFINE statement:

```
DEFINE 1 "Count"
...
ASSIGN %Count "10"
```

shows an example of this in practice.

Numeric variables

There is no predefined numeric variable type; if a script variable is referenced in a context where the contents must be numeric then the current contents of the variable will be evaluated and used numerically if possible. If the current contents cannot be evaluated as a numeric value then the OK status will be set false and it is up to the executing script to check whether or not the operation was successful or not.

Valid numeric values can be supplied either as decimal integers (with or without decimals) or in exponential format. Decimal integers should have at most 9 figures before the decimal point; values supplied in exponential format must be within the limits handled by the script language (the absolute value must be between 2.9E-39 and 1.7E+38).

Numeric results from script computations (ADD, SUBTRACT, MULTIPLY and DIVIDE) will be returned as straightforward numeric values (integers with trailing decimal zeros removed) if the absolute value of the result is between 0.01 and 32767. Otherwise the result will be returned in exponential format. The script TRUNCATE command can be used to convert numbers outside this range into numeric values.

NOTE
If an overflow condition occurs with a numeric value, then the script will terminate with the message “Value of numeric parameter out of range”.

Built-in variables

The script language provides for a number of 'built-in' variables that may be used in place of predefined strings or script variables. Note that these are 'read-only'; in other words, you can only use them in those places where they would not be modified (in general, in the same places as you can use literal strings).

The following built-in variables are supported:

`$BETA`	Single character specifying betatest release level (null for production releases)
`$CADDRESS`	For external interface (see separate chapter, page 211)
`$CALLER`	Calling address (for scripts that wait for incoming calls over X.25/NetBIOS)
`$CASE`	Returns the current case sensitivity setting (see `SET CASE`)
`$CERR`	Returns error position in failing `CALC` command
`$CFGCHANGED`	Returns 1 if modifications have been made to the current configuration, otherwise 0
`$CHANDLE`	Window handle of emulator child window
`$CMDLINE`	Returns the command line parameters used to start this copy of Glink
`$COLLECT`	Last (up to) 255 characters received while waiting in `ONLINE`, `RECEIVE` or `MATCH` statement
`$COMMENT`	Comment from dial directory
`$CONFIG`	Name of current configuration file
`$CRECEIVE`	Provides a count of the total number of characters received
`$CSEND`	Provides a count of the total number of characters sent
`$CTABS`	Returns 1 if the clipboard with tabs option is set, otherwise 0
`$CX`	Returns last column that was right-clicked
`$CY`	Returns last row that was right-clicked
`$DBLCLICK`	Contains ID of list or combo box that was double-clicked, if any
`$DBOX`	Returns the number of the button that was used to exit a dialog box
`$DDIR`	Returns the name of the current dial directory
`$DELIMITERS`	Returns the current word delimiters as set with `SET DELIMITERS` or in the screen options setup
`$DEVICE`	Returns the allocated LU name on TN3270E interfaces
`$DIAL`	Number of last dial entry (always 3 digits)
`$DIRECTORY`	Name of current directory

`$DOWNLOAD`	Name of Glink download directory
`$DRESULT`	Result string from last dial
`$DTYPE`	Number corresponding to current download type
`$DX`	X-coordinate last DBOX was left at
`$DY`	Y-coordinate last DBOX was left at
`$EMSG`	Current error message from status line/bar
`$ERRORLEVEL`	Returned error level from last DOS command
`$FE`	Configured editor name (`/FE`)
`$FIELD`	Current field number (VIP forms mode)
`$FILE`	Picked file name (see PICK command)
`$FL`	Configured file display program (`/FL`)
`$FNDX`	Used with `FIND`, returns column of string found
`$FNDY`	Used with `FIND`, returns row of string found
`$FORM`	Current TSM8/TCS form name
`$FP`	Configured file printer (`/FP`)
`$FPOS(#n)`	Current line number in file #n
`$FRAMEPAPER`	Current frame wallpaper file if any
`$FREE`	Free disk space on current drive (bytes)
`$FSEEK(#n)`	Current absolute position in file #n
`$FTP`	Current progress of FTP: 0=finished, 1=starting up, 2=running
`$FTPADDRESS`	Returns the current host IP address for FTP transfers
`$FTPCONFIG`	Returns the current host configuration filename for FTP transfers
`$FTPDEFAULT`	Returns the name of the default host name for FTP transfers
`$FTPERROR`	Returns an error message in the case of a failed FTP transfer
`$FTPHOST`	Returns the current host name for FTP transfers
`$FTPMODE`	Returns the current connection mode for FTP transfers (0 = Normal, 1 = PASV)
`$FTPPASSWORD`	Returns the current login password for FTP transfers
`$FTPRESULT`	Status for last FTP transfer (0=OK, else error code)
`$FTPSILENT`	Returns the current setting for FTP silent mode (0 = off, 1 = on)
`$FTPTRANSFER`	Returns the current transfer mode for FTP transfers (0 = Auto, 1= ASCII, 2 = BINARY, 3 = LOCAL8)
`$FTPUSER`	Returns the current login user name for FTP transfers
`$FX`	Current font size (X) in pixels
`$FY`	Current font size (Y) in pixels
`$Gn`	nth element of global parameter `$GPARAM`
`$GLCFG`	Name of Glink configuration directory
`$GLDEMO`	Name of Glink demo directory

`$GLINK`	Name of Glink base directory
`$GLSCR`	Name of Glink script directory
`$GLSUSE`	Name of Glink user script directory
`$GLUSE`	Name of Glink user directory
`$GPARAM`	Global parameter value (see `GPARAM`)
`$HINSTANCE`	Instance handle of Glink task
`$HMSG`	Current host message from status line/bar
`$HOST`	Current host machine name
`$IDLE`	Provides the current idle timer value (see `SET IDLE`)
`$IFILE`	Name of file that default icon is currently loaded from
`$INSTANCE`	Instance number
`$INTERFACE`	Zero if line interface is down, one if interface is up
`$INUMBER`	Number of default icon in `$IFILE` file
`$IPADDRESS`	Provides IP address for current host (Windows sockets only)
`$ISTATUS`	Contains the status of the last file operation that accessed the internet (3-digit numerical code followed by text)
`$KEYPRESS`	Last key pressed while in script
`$LANGUAGE`	Language key for program texts
`$LASTFILE`	Last file downloaded (or started downloading) with a protocol file transfer
`$LEVEL`	Current script stack depth (for debugging only)
`$LICBACKUP`	Glink license backup server name
`$LICENSE`	Full text of license info as displayed in text tab of the license upgrade dialog box
`$LICSERVER`	Glink license server name
`$LINE`	Current line of emulator screen
`$LNNO`	Current script source line number
`$LOCIPADDR`	IP address this machine used for last connection (see also $MYIPADDR, $MY4IPADDR, $MY6IPADDR, $MYAIPADDR)
`$MEM`	Available free memory (bytes)
`$MINIT`	Contents of modem init string
`$MOPTION`	Number of last selected option from a menu (zero if no option ever selected)
`$MRECT`	'1' if current mark is rectangular, otherwise zero
`$MSRECT`	'1' if current scrollback mark is rectangular, otherwise zero
`$MSX1`	Left coordinate of scrollback mark, zero if no mark
`$MSX2`	Right coordinate of scrollback mark, zero if no mark
`$MSY1`	Top coordinate of scrollback mark, zero if no mark
`$MSY2`	Bottom coordinate of scrollback mark, zero if no mark See the description of the `RSBK` command for an example of the use of the above four variables.

Variables

`$MWHEEL`	Returns the mouse wheel rotation value and resets that value to zero (each click gives 120 units)
`$MX`	Current mouse X-coordinate
`$MY`	Current mouse Y-coordinate
`$MX1`	Left coordinate of screen mark, zero if no mark
`$MX2`	Right coordinate of screen mark, zero if no mark
`$MY1`	Top coordinate of screen mark, zero if no mark
`$MY2`	Bottom coordinate of screen mark, zero if no mark
	See the description of the `RSCR` command for an example of the use of the above four variables.
`$MY4IPADDR`	IPv4 address for this machine (see also $LOCIPADDR, $MYIPADDR, $MY6IPADDR, $MYAIPADDR)
`$MY6IPADDR`	IPv6 address for this machine (see also $LOCIPADDR, $MYIPADDR, $MY4IPADDR, $MYAIPADDR)
`$MYAIPADDRE`	All IP addresses for this machine, comma separated (see also $LOCIPADDR, $MYIPADDR, $MY4IPADDR, $MY6IPADDR)
`$MYIPADDR`	IP address for this machine (see also $LOCIPADDR, $MY4IPADDR, $MY6IPADDR, $MYAIPADDR)
`$NAME`	Current user name
`$NCOMPUTER`	Network computer name
`$NGLINK`	Number of copies of Glink currently executing
`$NSESSION`	Number of active sessions
`$NT`	'1' if running under Windows Server OS, otherwise '0'
`$NUSER`	Network user name
`$OLEERROR`	Provides error code when OLE command fails (if available)
`$OLERESULT`	Provides error text when OLE command fails.
`$Pn`	nth parameter (see below)
`$PARAMETER`	Parameter string (see below)
`$PASSWORD`	Password from dial directory
`$PETX`	Returns the ETX position at the time of the most recent transmit command, in scrollback coordinates
`$PETY`	Returns the ETY position at the time of the most recent transmit command, in scrollback coordinates
`$PHONE`	Phone number of current dial entry
`$PKEY`	Returns the private key used in the SSH PuTTY interface
`$PRINTER`	Currently configured printer
`$PSTX`	Returns the STX position at the time of the most recent transmit command, in scrollback coordinates

`$PSTY`	Returns the STY position at the time of the most recent transmit command, in scrollback coordinates
`$RESOURCE`	Provides the current host profile name or TNVIP resource
`$RLTERM`	Provides the terminating character for the `RCVLINE` command
`$ROUND`	Current script rounding state (see `SET ROUND`)
`$SBKL`	Number of active scrollback lines
`$SCAN`	Offset of string found with `SCAN` command
`$SCRIPT`	Returns the fully qualified path name of the script currently running
`$SERIAL`	Glink serial number
`$SERVER`	Currently configured host or IP address
`$SESSION`	Current session number
`$SID`	Current session identifier
`$SSHPASS`	SSHD password
`$SSHSERVER`	SSHD server name
`$SSHUSER`	SSHD user name
`$STATUS`	Status string (described below)
`$STITLE`	Current scrollback window title
`$STX`	Returns column for start of transmission pointer (zero if none set)
`$STY`	Returns row for start of transmission pointer (zero if none set)
`$SWDX`	Width of scrollback window, in pixels
`$SWDY`	Height of scrollback window, in pixels
`$SWLVL`	Current script switch stack level
`$SWX`	X-coordinate of scrollback window, in pixels
`$SWY`	Y-coordinate of scrollback window, in pixels
`$SX`	Screen size (columns)
`$SY`	Screen size (rows)
`$TIMEOUT`	Current timeout value for script receives
`$TITLE`	Current window title
`$TURN`	Current turn status (1=have turn, 0=not)
`$UPLOAD`	Name of Glink upload directory
`$VERSION`	Glink version number
`$W32`	‘1’ if using 32-bit version of program, otherwise ‘0’
`$WALLPAPER`	Name of current wallpaper file (if any)
`$WDIR`	Path for Windows directory (note: no trailing `'\'` character)
`$WDL`	Used with `GETWORD`, returns length of word
`$WDX`	Width of main window, in pixels
`$WDX1`	Used with `GETWORD`, returns column for start of word
`$WDX2`	Used with `GETWORD`, returns column for end of word

$WDY	Height of main window, in pixels
$WDY1	Used with GETWORD, returns row for start of word
$WDY2	Used with GETWORD, returns row for end of word
$WHANDLE	Window handle of Glink main window
$WHEN	Pattern number for last activated WHEN
$WINDOW	Current window state: 0 = normal, 1 = maximized, 2 = minimized
$WINSTAMP	Returns the version of Windows for which this copy is stamped, in the same format as for $WINVER
$WINVER	Returns the current version of Windows. Windows Vista and Windows Server 2008 return 600, Windows 7 and Windows Server 2008 R2 return 601, Windows 8 and Windows Server 2012 return 602, Windows 8.1 and Windows Server 2012 R2 return 603, and Windows 10 returns 1000. See also $NT
$WORD	Used with GETWORD, returns selected word
$WSYSDIR	Path for Windows system directory. Note: no trailing '\'.
$WX	X-coordinate of main window, in pixels
$WY	Y-coordinate of main window, in pixels
$X	Current screen X coordinate
$Y	Current screen Y coordinate

Parameters (see $Pn and $PARAMETER) may be supplied for scripts either in the command line (using the /I parameter), this being used primarily for the startup script, or in the script command line. Starting a script using ALT+O may be done specifying both the name of a script and a number of parameters, separated with spaces. Internal calls to scripts may use the same facility, for example:

```
CALL "MYSCRIPT P1 P2 P3 P4"
```

In this case $PARAMETER will contain the string "P1 P2 P3 P4", while $P1 will contain "P1", $P2 will contain "P2", and so on.

The $STATUS variable

The `$STATUS` variable contains a number of things that have to do with the current emulator status. Not all of these are relevant for all modes, but all are returned with some value or another whether they are meaningful or not. You are therefore guaranteed that the position of each indicator inside the string returned will be independent of the particular emulation mode that is active. These positions are also guaranteed not to change on future versions of the emulator. Single-character positions representing values that are ON or OFF are returned as either Y or N. The following data may be found:

Pos	*Contents*
1-5	Abbreviation for current mode (VIP, V77, DKU, D7102, 3270, 5250, 3151, ANSI, VT220, VIEW, TTEL)
6	Echoplex mode
7	Insert mode
8	Roll mode
9	Local mode
10	Data capture active
11	Graphics mode
12	Auto LF mode
13	Keyboard locked
14	VIP mode (C=char, T=text, F=form)
15	TX-RET mode
16	Auto tabbing mode
17	Print logging active
18	VIP block mode active

Pattern variables

Another set of twenty variables is available. These are called pattern variables, and are used to 'monitor' the data coming from the host machine. Note that they are entirely separate from the normal variables mentioned above, and have their own special commands. To distinguish pattern variables from normal variables, they use a different prefix, the exclamation mark (!). Assignment to a pattern variable is done using the `PATTERN` command, like this:

```
PATTERN !1 "--more--"
```

Patterns may be referred to either using `IF` statements or `WHEN` statements. The `IF` can be used after any other statement (like `RECEIVE` or `MATCH`) that waits for input from the line to test whether or not the specified pattern was seen. The `WHEN` statement is used to specify some action that should be performed whenever the specified pattern is seen. See the description of these specific commands for more details. As for the normal variables, you may leave out the '!' prefix (in that it should be clear from the context that a pattern variable is being used) but we suggest that you use it for readability.

Also in the same way as for normal variables, indirection may be used, by replacing the '!' prefix with an underline character (_). When you do this the contents of the variable with the number you specify will be used as the number of the pattern to set. For example, the sequence of commands:

```
ASSIGN %3 "7"
PATTERN _3 "--more--"
WHEN _3 SEND " "
```

will set pattern number 7 to the value of `"--more--"` and in the same way the `WHEN` statement will refer to pattern 7, in that the `%3` variable contains the number 7.

File variables

A third type of variable is used for yet another set of commands, allowing you to perform various manipulations with files from your script. Up to 9 files may be referenced simultaneously, and again numbers are used to refer to these. In this case the number-sign (#) prefix is used to distinguish file variables from other types of variable. For example:

```
IF EOF #2 GOTO ENDPROC
```

directs the script to continue processing at the label `'ENDPROC'` if all the data on file number two has been processed. Detailed descriptions of the file commands will be found in later chapters.

The number-sign type of variable is also used to denote a DDE connection number. There may also be up to 9 of these, in addition to the 9 files.

Script command categories

Alphanumeric handling

`ADD`	adds two numbers
`CALC`	calculates the result of a complex formula
`CMPNUM`	compares two numbers
`COMPARE`	compares two strings
`CONCAT`	concatenates two strings
`DIVIDE`	divides two numbers
`EXTRACT`	extracts one parameter from a delimited string
`FILTER`	removes selected characters from a string
`FIX`	converts control chars to display format
`GETLENGTH`	gets length of a script variable
`LCASE`	converts a variable to lower case
`MULTIPLY`	multiplies two numbers
`REPLACE`	replaces all occurrences of one string with another
`SCAN`	checks for the presence of one string in another
`SPLIT`	splits a string by contents
`SUBRIGHT`	picks right-hand portion of string
`SUBSTR`	picks portion of a string
`SUBTRACT`	subtracts two numbers
`TRIM`	Trims leading and/or trailing spaces and controls
`TRUNCATE`	formats a number
`UCASE`	converts a variable to upper case

Compiler and debugging commands

`#ELSE`	introduces alternate compilation
`#ENDIF`	ends conditional compilation
`#IFDEF`	starts conditional compilation
`#IFNDEF`	starts conditional compilation
`DEBUG`	turns line debugging on or off
`DEFINE`	defines alternate name for script variable
`INCLUDE`	includes source code from another file

`OBJECT`	defines target file for compiled script
`STRACE`	turns debugging mode on or off
`TRACE`	traces line numbers

Configuration control

`CFIX`	is an expert command for config changes
`CFXW`	is an expert command for config changes
`CHANNEL`	sets logical communications channel
`CONFIG`	changes to another configuration
`CRDB`	reads a byte from configuration
`CRDW`	reads a word from the configuration
`CREAD`	reads several bytes from configuration
`CRLF`	switches added line feeds
`CTYPE`	sets comms interface type
`ECHO`	turns echoplex mode on and off
`EIGHTBIT`	sets 8-bit mode on and off
`LOCAL`	sets local mode on or off
`MDIAL`	sets modem dial string
`MINIT`	sets modem init string
`MODE`	switches between different emulations
`MONO`	sets monochrome mode on/off
`PACE`	sets line pacing
`PARITY`	sets the communications parity
`PORT`	sets the communications port
`ROLL`	toggles roll mode
`SET`	sets one of many different configuration options
`SETMACRO`	sets the value for a keyboard macro
`SPEED`	sets the communications line speed
`STRIP`	sets parity stripping on/off
`TSMDIR`	sets TSM8/TCS forms directory

Control structures

`*`	introduces a comment
`BEGIN`	starts a group of statements (`IF`)
`CALL`	executes another script as a subroutine
`CASE`	defines one alternative for 'switch' construct
`CHAIN`	gives control to another script
`CSWITCH`	is a `'C'` style switch statement
`DEFAULT`	defines default action in 'switch' construct
`ELSE`	is an alternative statement for `IF`
`ENDIF`	ends a group of statements (`IF`)
`ENDSWITCH`	defines end of 'switch' construct
`ENDWHILE`	defines end of 'while' construct
`EXITSWITCH`	exits from current 'switch'
`GOSUB`	performs a subroutine
`GOTO`	continues processing from another place in the script
`HALT`	stops the emulator
`IF`	introduces various forms for testing
`LABEL`	defines a place you can `GOTO` or `GOSUB`
`NEW`	Start a new script at top level
`POPUP`	removes last return from call stack
`QUIT`	stops the script
`RETCALL`	returns directly to last `CALL`
`RETURN`	returns from a `GOSUB` or a `CALL`
`SWITCH`	is a Pascal-style switch statement
`WHILE`	defines a program loop

Dial directory

`DCHANGE`	modifies an entry in the dial directory
`DFIND`	finds entry in the dial directory
`DIAL`	dials a number in the dial directory
`DMARK`	marks an entry in dial directory
`DREAD`	reads specified number from dial directory
`DUNMARK`	unmarks a number in the dial directory
`MANDIAL`	performs a manual dial
`RDIAL`	does a queued dial

File I/O commands

FCLOSE	closes a file
FLOC	returns the current line number in a text file
FOPEN	opens a file
FPOS	positions to a given line in a file
FRDBLOCK	reads a block from a file
FRDCHAR	reads a character from a file
FRDLINE	reads a line from a file
FSEEK	positions to a given location in a file
FSIZE	gets the size of a file in bytes
FSKIP	skips lines on a file
FWTBLOCK	writes a block to a file
FWTLINE	writes a line to a file
GPROFILE	reads parameters from INI files
PPROFILE	writes parameters to INI files

File transfer and control

BINARY	sets and resets the binary mode of Kermit or FTP
CAPTURE	toggles data capture mode
DOWNLOAD	sets download directory name
FTP	controls FTP transfers and related procedures
GETFILE	starts a file transfer from the host to the PC
LOG	turns print logging mode on and off
PUTFILE	starts a file transfer from the PC to a host
SERVER	sends a command to a Kermit server or starts a local Kermit server
UPLOAD	sets the upload directory

Host interaction commands

Command	Description
`BREAK`	sends a break
`CONNECT`	connects the physical communications line (use `NETCONNECT` for normal network interfaces)
`CONVERSE`	is a combined receive and sndline
`DISCONNECT`	disconnects the physical communications line (use `NETDISCONNECT` for network interfaces)
`DPATTERNS`	resets all patterns
`DWHENS`	deletes all active 'WHEN' statements
`FCODE`	sets a function code
`FLUSH`	empties the input buffer
`GWCONNECT`	makes a connection through a gateway
`ISOCONNECT`	initializes ISO connect menu
`LINE`	simulates line input
`MATCH`	waits for a pattern to arrive
`NETCONNECT`	connects to network host
`NETDISCONNECT`	disconnects from a network host
`ONLINE`	waits online
`PATTERN`	defines a pattern
`PREMOTE`	sets remote PAD parameters for X.25
`PSET`	sets PAD parameters for X.25
`RCVLINE`	receives from host into variable
`RCVTURN`	waits for turn (on interfaces that use turn)
`RECEIVE`	waits for a defined string from the host
`RECS`	waits for string (non-contiguous)
`SEND`	sends a message to the host with no terminator
`SNDLINE`	sends a message and a terminator to the host
`TRANSMIT`	sends a message without local emulation
`TRNLINE`	sends a terminated message without local emulation
`WHEN`	defines an action for when pattern is received

Key definitions and handling

`ABORT`	defines script abort key
`ENABLE`	defines script 'enable' key
`GETKEY`	gets one of a defined set of characters from keyboard
`KEYBOARD`	loads a transliteration file
`KEYKERMIT`	sets the Kermit transliteration file
`KEYS`	emulates a key sequence
`LAYOUT`	defines keyboard layout file
`ON`	does key reprogramming, error/timeout actions
`PAUSE`	defines pause key for script
`PERFORM`	performs internal emulator functions
`RKEY`	resets an `ON KEY` action for one key
`TCKEY`	(Atlantis V8 only) sends a TCU function key
`WKEY`	waits for any keypress

Menu handling

`DOMENU`	executes a user menu
`MENU`	defines start of a user menu
`MOK`	adds an `'OK'` button to a script menu
`MOP`	defines a menu line and action
`MOPC`	defines a menu line with activating key and action
`MOPTION`	sets the cursor position inside a menu
`MPOS`	defines where to display a menu on the screen
`MTEXT`	defines a menu text line
`NOMENU`	removes all active menus
`REMENU`	reactivates currently displayed menu
`UNMENU`	removes one active menu

NOTE
These script commands are left for backward compatibility. More specialized menu and dialog box commands are available in the Windows-specific list below.

Screen and cursor control

BEEP	sounds the alarm
CLEAR	clears the screen
CURSOR	turns cursor on and off, or changes its shape
DSCREEN	dumps screen to file
EMULATE	emulates a string locally
FIND	Finds a specified string on the screen
GETWORD	collects a word from the main or scrollback screen
HOST	inserts host name in status line
MESSAGE	sends a message to the terminal
NAME	inserts user name in status line
POS	moves the cursor to a set position
RATR	reads attributes from a position on the screen
RFORM	reads a field from a form
RSBK	reads characters from the scrollback buffer
RSCR	reads characters from screen
SCREEN	switches screen updates on or off
SECURE	removes access to potentially sensitive data
SHOW	shows a message locally, no CRLF
WELCOME	displays the welcome menu

System-related commands

APPEND	appends one file to another
CD	changes the working directory
COPY	copies one file to another
DOS	executes a program
DOSN	executes a program
ERASE	erases a file
EXECUTE	executes a program
EXISTS	tests for existence of file or directory
FNEXT	finds next matching file
FSEARCH	finds first matching file
MD	creates a directory
PRINT	prints the specified string
RD	deletes a directory
REN	renames a file
UVTI	starts a UVTI shell

Timing commands

`DELAY`	waits a number of seconds
`DTENTHS`	delays, in tenths of a second
`DTIME`	delays until a specified time
`ERRORGOTO`	defines a place to `GOTO` when errors occur
`IDLE`	waits for host to remain idle for a defined period
`RESET`	resets error/timeout actions
`TIMEOUT`	defines how long a `RECEIVE` should wait

User input commands

`INFILE`	collects file name from the user
`INPC`	collects user input, cursor placed at end of input
`INPUT`	collects user input
`INVISIBLE`	collects user input, echoing asterisks or question marks
`PICK`	picks a file using the file display

Variable handling commands

`ASSIGN`	puts a value in a variable
`CXRESTORE`	restores a script context
`CXSAVE`	saves a script context
`DVARIABLES`	resets all variables
`GETDATE`	formats and places today's date into variable
`GETENV`	gets contents of environment string
`GETLENGTH`	obtains length of a script variable
`GETTIME`	formats and places time of day into variable
`GETVALUE`	converts from binary format
`GPARAM`	sets the global parameter value
`PARAM`	sets the script parameter

Windows-specific commands

`ACTIVATE`	passes control to another Windows application
`ADDMENU`	adds a user entry to the Windows menu bar
`ADMSHELL`	executes a command as administrator/alternate user
`BUILDMENU`	defines a new entry in the Windows menu bar or adds a submenu to the current menu
`BUTTON`	defines a button for the button bar
`CONTEXT`	Adds a user entry to the context (right mouse button) menu
`DBOX`	starts a dialog box definition
`DDEADVISE`	asks for notification of changed data
`DDECLOSE`	closes a DDE connection
`DDEEXECUTE`	executes a DDE command
`DDENAME`	sets the DDE application name
`DDEOPEN`	opens a DDE connection
`DDEPOKE`	pokes a DDE value
`DDEREQUEST`	requests a DDE data item
`DELMENU`	deletes a user menu bar item
`ENDBUILD`	ends a Windows submenu defined with the `BUILDMENU` command.
`FNDEXEC`	Provides the name of the associated executable
`FVERSION`	Extracts the program version number
`ICON`	defines default icon to display when minimized
`ISSERVICE`	Used to specify that Glink is running as a service
`MBAR`	enables or disables items on the menu bar
`MCURSOR`	sets mouse cursor shape
`MFONT`	selects the font to use when displaying menus
`MOVEWINDOW`	moves and/or resizes the emulator window
`MVSCROLL`	moves and/or resizes the scrollback window
`MSGBOX`	displays a message in a separate window
`OEM`	specifies that script is in OEM character set
`OLE`	provides OLE automation controller functionality
`PLAY`	plays a waveform file
`SEPMENU`	adds a horizontal separator to a user menu
`SHELL`	executes a command or associated command
`STITLE`	sets the caption for the scrollback window
`TITLE`	sets the Windows caption
`URLSHOW`	invokes your browser to display a URL
`WINDOW`	defines how the main screen should be displayed

Commands for backwards compatibility

The following commands are supported for compatibility with previous releases of the software:

`ADIAL`	dials a given entry only once
`GETNAME`	gets login name from dial directory
`GETPASS`	gets password from dial directory
`GETX`	gets screen X coordinate
`GETY`	gets screen Y coordinate

Script commands

In this chapter, the following conventions are used in the description of the syntax of the different script commands:

<... > enclose syntactical elements

{...|...} enclose lists of alternative options

[...] enclose optional items

[...[...]] enclose variable length lists of optional items

<%var> is used where use of a script variable is mandatory

<#file id> is used where use of a file identifier is mandatory

<!pattern> is used where use of a pattern number is mandatory

<#dde> is used where use of a DDE connection number is mandatory

The #ELSE directive

Syntax: #ELSE

The #ELSE directive introduces a set of script commands that will either be compiled or ignored, depending upon the previous #IFDEF or #IFNDEF directive.

The #ENDIF directive

Syntax: #ENDIF

The #ENDIF directive marks the end of a group of script commands that have been conditionally compiled as a result of a previous #IFDEF or #IFNDEF directive.

The #IFDEF directive

Syntax: `#IFDEF {i.j.k[x]|$VAR|DOS|WINDOWS|MAC}`

The `#IFDEF` directive introduces a group of script commands that will either be compiled or ignored, depending upon the argument that you specify. The following may be used as arguments to the `#IFDEF` directive:

`i.j.k`	True for all releases from i.j.k and upwards
`i.j.kx`	True for all releases from i.j.k beta x and upwards
$var	True if $var is a valid internal variable
`DOS`	True for the DOS version
`MAC`	True for the Macintosh version
`WINDOWS`	True for the Windows versions
`WIN32`	True for the Windows 32-bit version

The group of commands that should be compiled or ignored is terminated with an `#ENDIF` directive. The group may also optionally contain an `#ELSE` directive, thus allowing you to compile one of two different sets of commands, depending upon the Glink release or the version of the emulator being used. `#IFDEF` directives may be nested to any level.

For example, to include a group of statements that should only be compiled for a Win32 platform, and only if the version of Glink being used is at least 7.1.0, you could use:

```
#IFDEF WIN32
#IFDEF 7.1.0
... script statements ...
#ENDIF
#ENDIF
```

See also: `#ELSE`, `#ENDIF`, `#IFNDEF`.

The #IFNDEF directive

Syntax: `#IFNDEF {i.j.k|DOS|WINDOWS|MAC}`

The `#IFNDEF` directive introduces a group of script commands that will be compiled or ignored depending upon whether the condition named is true or false (in the opposite sense to `#IFDEF`). The commands following `#IFNDEF` are only compiled if the condition is not true.

The ABORT command

Syntax: `ABORT {<keyname>|NONE}`

A script that is executing may normally be terminated at any time by pressing the `Esc` key. This may not always be desirable, and the `ABORT` command allows you to move this function to any other key, or indeed disable the function altogether (this last for situations where you wish to avoid the user being able to terminate the script you are writing).

For a list of valid keynames, refer to the `KEYS` command. Note that the `ABORT` key has an effect not only for the current script, but also for all subsequent scripts until another `ABORT` command is executed. In other words, if you specifically want the user to be able to abort the script you are writing using the `Esc` key then you should include:

```
ABORT Esc
```

somewhere at the start of your script.

The ACTIVATE command

Syntax: `ACTIVATE <window>`

This command allows you to pass control to another window in the Windows environment. The parameter you provide should specify the title of the window to which you wish to pass control, for example:

```
ACTIVATE "Notepad - MESSAGE.TXT"
```

For advanced users, you may also activate a window based on the Windows class name of the window you wish to activate. Do this by using a '#' prefix in front of the class name, for example:

```
ACTIVATE "#Progman"
```

On Win98 and above some programs may not accept receiving the input focus.

The ADD command

Syntax: `ADD <%var> <number>`

The ADD command allows you to compute the sum of two numbers. The first parameter must be a script variable, while the second may be a script variable or a constant. The result of adding the two numbers is placed in the script variable specified first. For example:

```
ADD %3 1
```

adds 1 to the present contents of the %3 variable, leaving the result in %3. Note that the result may be stored in exponential format to keep maximum precision. If you need to print a result that may be outside the range 0.01 to 32767, you can use the TRUNCATE command to format the number in a more suitable way. If the addition can be performed correctly then the OK variable is set true. If not (because one of the two operands was non-numeric) then it's set false.

The ADDMENU command

Syntax: `ADDMENU <scriptname> <string>`

The ADDMENU command allows you to add a new entry to the menu bar. The first parameter must be the name of a script file that will be executed when the menu item is chosen, while the second is the text that should be displayed in the menu. The first time such a command is executed, a new entry is added to the main menu bar, which is given the name "User" (this text is defined in the Glink string resources and may be changed by editing the resources if desired). The text may define a shortcut key in the usual way by prefixing the shortcut letter with an ampersand (&). For example:

```
ADDMENU "LOGIN.SCR" "&Log into host"
```

would add an entry called 'Log into host' with a shortcut 'L'. When this entry is selected by the user, Glink will attempt to execute the script called `LOGIN.SCR`.

For more advanced use of the menu bar, see the `BUILDMENU` command on page 37. If you wish to add horizontal separating lines between some of the items defined with `ADDMENU` this may be done using the `SEPMENU` command.

The user-defined menu may be deleted at any time using the `DELMENU` command.

The ADMSHELL command

Syntax: `ADMSHELL <command/file>`

This command executes an application, or alternatively the application that is associated with the file type of the specified file. It will also invoke ‘run as’ functionality so that you may specify an administrator on whose behalf the application will run, otherwise it is equivalent to the `SHELL` command. Typically you will be allowed to enter login information in a popup window for the user that will run the command.

The APPEND command

Syntax: `APPEND <source name> <destination name>`

This command allows you to append the contents of one file to another without the inconvenience of having to use an external utility. If the destination does not exist then the source file will simply be copied.

The ASSIGN command

Syntax: `ASSIGN <%var> <string>`

Glink scripts allow you to define up to 99 different 'variables', numbered from 1 to 99. These may be used anywhere where you could have used a text string. To assign a value to variable number 13 you could for example use a command like:

```
ASSIGN %13 "Hello, how are you?"
```

You can then send this string by using the variable number with a percent sign in front. For example:

```
SEND %13
```

would send the string we just defined.

The BEEP command

Syntax: `BEEP`

This command simply sounds the local alarm (unless you have set the option in the configuration menus, which disables it). No parameters are needed.

The BEGIN command

Syntax: `BEGIN`

This is used to mark the start of a group of statements that are to be executed following an `IF` statement. Normally only the first statement following the `IF` statement will be executed if the test succeeds, but use of a group of statements delimited with `BEGIN` and `ENDIF` overrides this. For example:

```
IF (%8 EQ "X") BEGIN
  MESSAGE "Please wait..."
  GOSUB Sub1
ENDIF
```

The BINARY command

Syntax: `BINARY {ON|OFF}`

This command sets the transfer mode for a Kermit orFTP transfer. `BINARY ON` will set the next transfer into binary mode, while `BINARY OFF` will set it to text mode.

The BREAK command

Syntax: `BREAK`

This command will send a break signal to the host machine.

The BUILDMENU command

Syntax: `BUILDMENU <string>`

This command extends the functionality provided by the `ADDMENU` command. It allows both placing of additional entries into the main menu bar and building of submenus connected to items in the top-level menus you have defined. A `BUILDMENU` used at the top level will cause a new entry to be added to the main menu, while a `BUILDMENU` enclosed inside a `BUILDMENU / ENDBUILD` pair defines a submenu. For example:

```
BUILDMENU "MyMenu"
  ADDMENU "SCRIPT1" "First"
  BUILDMENU "Second"
    ADDMENU "SCRIPT21" "Second/1"
    ADDMENU "SCRIPT22" "Second/2"
  ENDBUILD
  ADDMENU "SCRIPT3" "Third"
ENDBUILD
```

will add a new entry in the menu bar called MyMenu. The menu will consist of three items, First, Second and Third. The First and Third items will execute `SCRIPT1` and `SCRIPT3` directly, while the Second item will produce a submenu with two selections for executing `SCRIPT21` and `SCRIPT22` respectively. A new group starting with another `BUILDMENU` immediately after this one would now add a separate item to the main menu bar.

There is a limit of nine levels of nesting of menus, and an absolute limit of 50 menu entries for all defined menus.

The BUTTON command

Syntax: `BUTTON <number> <keystroke> <string>`

This command defines a button for the button bar at the bottom of the screen, with the specified `<number>` (48 maximum). If the number is larger than the number of buttons currently being displayed then the button bar will be expanded to this number of buttons. If the command makes it impossible to display all texts in the currently displayed buttons, then an additional row will be displayed to make room for the text (up to a maximum of four).

`<keystroke>` can be any valid keystroke as defined in the explanation of the script `KEYS` command. For example, to define a button with the same function as the `F1` function key you would use:

```
BUTTON 1 F1 "F1"
```

Additionally you can use the format `MACRO-n` to associate the button with macro number 'n'. This allows you to define specific actions for buttons as opposed to duplicating the action of a particular key, using something like:

```
SETMACRO 21 "pwd^!"
BUTTON 1 MACRO-21 "PWD"
```

This type of definition can be extended (for example) to running scripts:

```
SETMACRO 22 "^*LOGIN.SCR"
BUTTON 2 MACRO-22 "Login"
```

You can set the total number of buttons specifically using the command:

```
SET BUTTON NUMBER number
```

Setting the number of buttons to zero will remove the button bar altogether. If you set the number of buttons to less than the number of rows currently being displayed then the number of rows displayed will be reduced correspondingly.

You can also set the number of rows of buttons to display using the command:

```
SET BUTTON ROWS number
```

(This command will not be respected if it would result in inability to display all texts in the buttons).

If you wish to associate a help text with the button in the same way as the predefined help texts for the toolbar, then you may do this using:

```
SET BUTTON HELP number "text"
```

The text will be displayed at the left-hand end of the status bar, and also in a small popup help window, if such windows are enabled. The same text will be displayed in each of these. If you wish to use different texts then specify these with a colon separating the two. For example:

```
SET BUTTON HELP 3 "Logout:Log out of the host machine"
```

This command will provide the text "Logout" in the popup help window, but show the longer text in the status bar. You may also change the font used to display the text in the buttons with the SET BUTTON FONT command, for example:

```
SET BUTTON FONT "Courier New Bold" 16
```

The CALC command

Syntax: `CALC %n "formula"`

This will evaluate the specified formula and place the result in the `%n` variable. The formula is evaluated using normal algebraic rules (exponentiation first, then multiplication and division, then addition and subtraction) and parentheses may be used to override those rules. The full list of available operators is as follows:

`a + b`	(addition)
`a - b`	(subtraction)
`a * b`	(multiplication)
`a / b`	(division)
`a ^ b`	(exponentiation)
`a \ b`	(modulus)
`a | b`	(logical OR)
`a & b`	(logical AND)
`a # b`	(logical XOR)
`a > b`	(logical shift right)
`a < b`	(logical shift left)
`!a`	(logical NOT)

All logical operators carry with them an implicit truncation operation if the terms involved are non-integral. Remember that the `^` sign has a special meaning in scripts so you will need to write `^^` as your exponential operator in practice. A number of standard functions are also supplied:

`ABS(x)`	returns the absolute value of x
`ARCTAN(x)`	returns the arctangent of x
`COS(x)`	returns the cosine of x (x specified in radians)
`EXP(x)`	returns the exponential of x
`FRAC(x)`	returns the fractional part of x
`LN(x)`	returns the natural logarithm of x
`LOG(x)`	returns the logarithm base 10 of x
`PI`	returns the value of pi
`RAND(x)`	returns a random number less than x
`ROUND(x)`	rounds x to the nearest integer
`SIN(x)`	returns the sine of x (x specified in radians)
`SQRT(x)`	returns the square root of x
`TRUNC(x)`	returns the integer part of x

If an error should occur during the evaluation of a formula then the script OK variable will be set to FALSE and you will also find the location of the error (offset into the formula) in the built-in $CERR variable. Errors in CALC commands will in general not cause a script to halt when such errors occur.

Variables specified using the '%' and/or '_' prefixes may be included directly in the formula so long as the numeric variable identifier is used. If you wish to use a DEFINEd value for the variable identifier then you must use parentheses to separate the elements of the formula. For example:

```
CALC %1 "%2+%3"
```

is acceptable, but

```
CALC %a "%x*%y"
```

is not, and must be written as:

```
CALC %a (%x "*" %y)
```

(For the technically inclined, this is because the % sign is treated as a unary operator by the expression evaluation mechanism, and therefore needs a numeric argument. While this precludes the use of compile-time constants, it does open possibilities for other constructions - for example something like "%(trunc(rand(20))+1)")

The CALL command

Syntax: CALL <scriptname>[!label]

This command will transfer control to another script file. However, Glink will remember where you were in your original file and when the CALLed script executes a RETURN statement, processing will continue from the next command after the CALL. The CALL command needs one item of information, the name of the script to give control to:

```
CALL "SCRIPT2.SCR"
```

The whole pathname must be provided (except the `'SCR'` extension, which is optional). If the script you are calling is in source format, you may additionally call it using the following format:

```
CALL "SCRIPT3.SCR!label"
```

In this case, execution of the script will commence from the label you have specified rather than from the beginning of the script. Otherwise note that all normal conventions that may be used when starting a script apply; see the section on *Starting a script* earlier in this manual, on page 2.

If the name of the script file to be called contains embedded spaces then you will need to supply an extra set of quotes around the name:

```
CALL "'MY SCRIPT2.SCRGL'"
```

This also allows a distinction between calling a script with parameters and calling a script with embedded spaces in the file name, or indeed doing both at the same time:

```
CALL "'MY SCRIPT2.SCRGL' param-1 param-2"
```

The CAPTURE command

Syntax: `CAPTURE {ON|OFF}`

This command toggles capture mode (same as the interactive `ALT+V` command). Captured output is placed in the current capture file. The name of this file may be changed using the initial command `'GETFILE ASCII filename'` to start the first capture.

The CASE command

Syntax: `CASE <string>`

The `CASE` command is used to specify one possible alternative inside a `SWITCH` or `CSWITCH` construct. See the documentation for `SWITCH` and `CSWITCH` for details. It takes a single parameter, providing the string with which to compare. For example:

```
CASE "1"
```

This specifies that the following statements should be executed for any variable specified in the SWITCH or CSWITCH statement that **starts** with the character '1'. If you want to check for **exactly** '1' then specify "1 " with a trailing space in the CASE statement.

The CD command

Syntax: CD <directory-name>

The CD command allows you to change the working directory without having to use the command. IF OK may be used to test whether or not the directory change was successful. For example:

```
CD "\WORK"
```

Note that many of the internal variables representing directories include a trailing backslash (\) so as to let you execute commands including those directories as prefixes to file names. For example, a construction such as ($DOWNLOAD "FILENAME.EXT") which works equally well whether the download directory is set or not. To use such directories with the CD command remember that you can use the command convention of '.' to represent the current directory:

```
CD ($DOWNLOAD ".")
```

This again works equally well whether or not the download directory is actually set.

The CFIX command

Syntax: `CFIX <location> <byte value>`

This command may be used to change various parameters in the configuration information (in memory) that are not otherwise easy to change using the normal script commands. `<location>` is the offset into the configuration information and `<byte value>` is a single byte that is to be placed there. The format of the configuration file is provided in the *Configuration file format* appendix to this guide, on page 227. Care should be exercised in changing the contents of the configuration file in that invalid contents in some of the fields may produce unpredictable results. Note also that changing the contents of the configuration file will in many cases not have an immediate effect but will only be acted upon if the configuration information is saved and reloaded.

The CFXW command

Syntax: `CFXW <location> <word value>`

This command is exactly the same as the `CFIX` command, but is used for changing values in the configuration file that are stored using two bytes rather than one.

The CHAIN command

Syntax: `CHAIN <scriptname>[!label]`

This command also allows you to give control to another script file, but unlike the `CALL` command, control will not be returned to the original script. It is however perfectly `OK` to do a `CALL` from script one to script two, a `CHAIN` from script two to script three, and then a `RETURN` from script three. This will take you back to the original `CALL` in script one. The `CHAIN` command looks just like the `CALL` command:

```
CHAIN "SCRIPT4.SCR"
```

Otherwise note that all normal conventions that may be used when starting a script apply; see the section *'Starting a script'* on page 2. See the script CALL command for examples of using script names with embedded spaces and/or using additional parameters.

The CHANNEL command

Syntax: `CHANNEL <number>`

This command allows you to set the logical communications channel or port for those network interfaces that use one. This includes the Eicon NABIOS, PC-NFS/Wollongong and Ungermann-Bass (interrupt 6B) interfaces.

The CLEAR command

Syntax: `CLEAR`

This command allows you to clear the screen without sending any data to the other end of the communications line.

The CMPNUM command

Syntax: `CMPNUM <string1> <string2>`

This command compares the values contained in two strings, both of which are interpreted as numeric values. If the strings do not both contain numeric values then the `OK` variable is set false (see the `IF` command) and the comparison performed as though the invalid value or values contained zero. If they both contain valid numbers, then the comparison is done and the result saved so that you may use the `IF` command to test what happened. For example, to check whether the value in script variable number 5 is greater than 10, you could use something like:

```
CMPNUM %5 "10"
IF GT GOTO GREATER
```

Note that such comparisons may be included directly in `IF` statements, see the description of the `IF` command (page 92) for more information.

The COMPARE command

Syntax: COMPARE <string1> <string2>

Similar to the CMPNUM command, this command allows you to make a comparison between any two strings, but interpreted with their ASCII values rather than as numbers (in other words, "100" would be less than "20"). Note that such comparisons may be included directly in IF statements, see the description of the IF command on page 92 for more information.

The CONCAT command

Syntax: CONCAT <%var> <string>

This command will concatenate a string to a variable, thus allowing you to build longer strings from smaller components. The first parameter must be one of the script variables, while the second may be either a literal or a script variable. Examples:

```
CONCAT %1 "^M"
```

adds a CR character to the present value of variable 1.

```
CONCAT %1 %5
```

adds the current value of variable 5 to the end of whatever is in variable 1. Note that when you just need to use the result of concatenating two strings in another command, this can be done directly, for example:

```
ERASE ($GLUSE "WORKFILE")
```

This erases the file named WORKFILE in the Glink user directory, and saves you using the CONCAT command to 'glue together' the two parts of the full pathname you want to use for the ERASE command.

The CONFIG command

Syntax: `CONFIG {<configuration filename>|SAVE}`

This command allows you to switch to another configuration file. The configuration file must be contained in the Glink user directory, and the full name (not just the extension) must be supplied. For example:

```
CONFIG "GLINK.CF1"
```

The `CONFIG SAVE` variant may be used to save the current configuration back to disk.

If the configuration cannot be loaded then the emulator will reset to the default options that were delivered with the system, and the script `OK` variable will be set false. Note that the `$CONFIG` internal variable may be used to save the current configuration name before you load a new setup, so that you may restore the current configuration if the load should fail:

```
ASSIGN %1 $CONFIG
CONFIG "NEW.glinkconfig"
IF NOT OK CONFIG %1
```

The CONNECT (modem) command

Syntax: `CONNECT`

This command 'connects' the communications line by raising the DTR signal. It would normally only be used in a situation where there was a preceding `DISCONNECT` command. Note that to connect on normal network interfaces you should use the `NETCONNECT` command.

The CONTEXT command

Syntax:

CONTEXT [MAIN | SCROLLBACK] ADD <script> <string>
CONTEXT [MAIN | SCROLLBACK] DELETE <string>
CONTEXT [MAIN | SCROLLBACK] SEPARATOR

The `CONTEXT` command allows you to use new entries in the context menus (the menus displayed when the right mouse button is pressed) in the main and scroll-back windows. `<script>` must be the name of a script file that will be executed when the menu item is chosen, while `<text>` is the text that should be displayed in the menu itself. The entry will be added at the end of the menu. The text may define a shortcut key in the usual way by prefixing the shortcut letter with an ampersand (&). For example:

```
CONTEXT MAIN ADD "LOGIN.SCR" "&Log into host"
```

would add an entry called 'Log into host' with a shortcut `'L'`. When this entry is selected by the user, Glink will attempt to execute the script called `LOGIN.SCR`. Note that if you need to use the actual screen position that was clicked with the mouse then its coordinates are available in the `$CX` and `$CY` built-in variables.

If you wish to add horizontal separating lines between some of the items in the context menu this may be done with the command:

```
CONTEXT MAIN SEPARATOR
```

Any of the entries in the menu may be deleted at any time using the command:

```
CONTEXT MAIN DELETE "text"
```

This will delete the entry that uses text in its entry (this may also be used to delete entries from the standard menu if you should choose to do so). If the entry deleted is immediately preceded by a separator then the separator will also be removed.

All of the above examples may be used to manipulate the scrollback context menu rather than the main context menu, simply by replacing the keyword MAIN with the keyword `SCROLLBACK`. For example:

```
CONTEXT SCROLLBACK ADD "PWORD.SCR" "Paste to &Word"
```

The CONVERSE command

Syntax: `CONVERSE <receive string> <send string>`

This command is a combination of the `RECEIVE` and `SNDLINE` commands. Two strings are provided, and the script will wait for the first one to arrive, then transmit the second. For example:

```
CONVERSE "Name? " "Mike"
```

Note that the second string is sent with `SNDLINE`, not `SEND`. Timeouts are effective in just the same way as for a normal receive while the emulator is waiting for the first string to arrive. Also, if a `SET IDLE` command is active then Glink will wait for the line to become idle for the specified time before sending the reply (this may be found necessary on half-duplex lines).

The COPY command

Syntax: `COPY <source name> <destination name>`

This command allows you to copy the contents of one file to another without the inconvenience of having to use an external utility. `IF OK` may be used to test whether or not the copy operation was successful.

The CRDB command

Syntax: `CRDB <%var> <location>`

This command can be used to read a byte from the configuration file into a script variable. This may be used in specialized applications that need to check the current status of some configuration option that is not available from the script language in any other way. The byte is inserted into the script variable in decimal format. See the *Configuration file format* appendix to this guide (page 227) for more information and examples of usage of the `CRDB` command. Note that the CRDW script command may be used to read data that is stored in the configuration file as a double-byte value.

The CRDW command

Syntax: `CRDW <%var> <location>`

This command can be used to read a word stored in the configuration file as two bytes into a script variable. This may be used in specialized applications that need to check the current status of some configuration option that is not available from the script language in any other way. The word is inserted into the script variable in decimal format. See the *Configuration file format* appendix to this guide (page 227) for more information and examples of usage of the `CRDB` command.

The CREAD command

Syntax: `CREAD <%var> <location> <length>`

This command can be used to read a byte or sequence of bytes from the configuration file into a script variable. This may be used in specialized applications that need to check the current status of some configuration option that is not available from the script language in any other way. Note that if the command is used to read bytes that are encoded in the configuration as binary values, then this is the way they will be provided in the `%n` variable. If a numeric value corresponding to a particular byte is what you need then `CRDB` should be used instead. The format of the configuration file and some script examples are provided in the *Configuration file format* appendix to this guide, on page 227.

The CRLF command

Syntax: `CRLF {ON|OFF}`

This is equivalent to the `Auto LF In` option in the emulator setup menu. When you use `CRLF ON`, the emulator will add linefeed characters to all CR characters received from the line. With `CRLF OFF` it does not. The option should be used in cases where the host appears to be writing data on the same line all the time.

The CSWITCH command

Syntax: `CSWITCH <string>`

The `CSWITCH` command marks the start of a switch construct. These are used to test the contents of a script variable (or built-in variable) for one of several alternatives. A basic example is provided in the description of the `SWITCH` command, and applies equally to `CSWITCH`. The only difference between `SWITCH` and `CSWITCH` is the way that the `CASE` statements are handled. In a `SWITCH` construct, as soon as a matching CASE has been found and executed, control passes to the `ENDSWITCH` statement, while in a `CSWITCH` construct, this happens only when an `EXITSWITCH` is used specifically. Even when a matching `CASE` is found, execution proceeds through the remaining `CASE` statements until either `ENDSWITCH` or `EXITSWITCH` is found. This means that `EXITSWITCH` must be provided specifically for those `CASE`s that are to be handled in the same way as for `SWITCH`.

An example will make this clear:

```
CSWITCH %1
  CASE "A"; MESSAGE "This is only executed for A"
  CASE "B"; MESSAGE "This is done for A and B"
  CASE "C"; MESSAGE "And this for A, B and C"
  EXITSWITCH;
  CASE "D"; MESSAGE "While this is done for D only"
ENDSWITCH
```

The CTYPE command

Syntax: `CTYPE <interface name>`

This command allows you to set the communications interface to be used. Allowable options here are:

`AV8`	Atlantis Bull TSA V8 DLL
`ETGX`	Eicon NABIOS / TGX
`DGA`	G&R / DGA (Direct GCOS Access)
`FPX`	Cirel FPX / VTI
`NABIOS`	Eicon NABIOS
`NETBIOS`	NetBIOS (raw)
`NETGAR`	NetBIOS (G&R)

NONE	No interface defined
PUTTY	PuTTY SSHD
SH8	Atlantis X.25 V8
SPX	SPX/IPX modem server
TAPI	Windows Telephony
VTI	Cirel FPX /VTI via VTI3.DLL
WINDOWS	Windows serial port access
WINSOCK	All TCP/IP protocols, see also `SET TCP <protocol>`

Note that when the `CTYPE` command is used it is the script's responsibility to disconnect from whatever interface was previously being used, and to reconnect using the new one, normally with the `NETDISCONNECT` and `NETCONNECT` commands.

The CURSOR command

Syntax: `CURSOR {ON|OFF|LINE|BLINK|BLOCK}`

Normally the cursor is left on when a script executes. In situations where speed is of the essence, some savings may be made by suppressing the extra overhead required to update the cursor. The `CURSOR OFF` and `CURSOR ON` commands allow you to do this. You may also use the `CURSOR` command to change the shape of the visible cursor in the same way as in the screen setup menu.

The CXRESTORE command

Syntax: `CXRESTORE`

This command restores a context previously saved using the `CXSAVE` command; what actually will be restored will depend upon the option used on the `CXSAVE` command. See the `CXSAVE` command below for details.

The CXSAVE command

Syntax: `CXSAVE [<number>]`

This command may be used to save script patterns and associated `WHEN` actions, some internal settings, and additionally some or all of the normal script variables. Normal use of this will be for utility scripts that might be called from other scripts where it is undesirable to disturb the contents of patterns and/or variables. If `<number>` is not specified then script patterns and active `WHEN` statements will be saved; if `<number>` is specified then all script variables starting with variable `n` will be saved as well. In both cases the current status of script rounding, case sensitivity and idle timing will be saved (see `SET ROUND`, `SET CASE` and `SET IDLE`). `CXSAVE 1` will for example save ALL script variables, `CXSAVE 4` all variables with the exception of the first three. The context thus saved is restored using the `CXRESTORE` command, where all saved data is returned to its original status. `CXSAVE 4` could be used for example in a case where the called script might need to return three values, which could then be left in `%1`, `%2` and `%3` without being disturbed by the subsequent `CXRESTORE`.

`CXSAVE` requires that enough free memory be available to hold the data being saved.

The DBOX command

Syntax: see below

The `DBOX` command allows you to define your own Windows dialog boxes directly from a script file, using most of the graphic controls supported by Windows, such as check boxes, buttons, radio buttons and list boxes.

In that the `DBOX` command is fairly complex, the complete description of the command has been included in a separate chapter of this manual, on page 177. Here we will just summarize the general format of the command for reference:

```
DBOX           X Y W H ["Caption"]
AUTOGROUP              "Text"                     [options]
CHECKBOX       X Y W H "Text"   %N                [options]
COMBOBOX       X Y W H          %N                [options]
CTEXT          X Y W H "Text"                     [options]
DEFPUSHBUTTON  X Y W H "Text"      number         [options]
EDITTEXT       X Y W H          %N                [options]
```

```
ENDGROUP
ENDHGROUP
ENDVGROUP
GROUPBOX       X Y W H "Text"                [options]
HGROUP         X Y W H
IBUTTON        X Y W H "Text"       number   [options]
ICON           X Y W H "Text"
LISTBOX        X Y W H          %N           [options]
LTEXT          X Y W H "Text"                [options]
PUSHBUTTON     X Y W H "Text"       number   [options]
RADIOBUTTON    X Y W H "Text"   %N  number   [options]
SIZEBUTTON     X Y W H "Text"       DW DH    [options]
RTEXT          X Y W H "Text"                [options]
VGROUP
ENDDBOX ["HelpFile"]
```

The DCHANGE command

Syntax: `DCHANGE <item name> {<string>|<option>}`

This command allows you to modify selected fields in the dial directory from a script procedure. The entry to be modified must be current (i.e. it must be the one to which you dialled most recently or one that you have positioned to using `DFIND` or `DREAD`). The following may be used for `<item name>`:

`APHONE`	The alternate number to use for this system
`COMMENT`	The comment displayed for this entry
`EXTRA`	The additional modem command to use when calling this system
`HOSTNAME`	The name of the host machine
`KEYBOARD`	The keyboard transliteration file to use
`LAYOUT`	The keyboard layout file to use
`LOGIN`	The login name to use on this system
`MODE`	The emulation mode to use (use the same names as used in the `MODE` command)
`PARITY`	The parity setting to use when calling this system (use the same options as used in the `PARITY` command)
`PASSWORD`	The password to use on this system
`PHONE`	The phone number to use for this system
`SCRIPT`	The name of the script file to use when calling
`STRIP`	Whether to strip parity or not (specify `YES` or `NO`)

The DDEADVISE command

Syntax: `DDEADVISE <#dde> <item name>`

This command is used to ask for notification of changed data on a DDE connection opened with the `DDEOPEN` command. Notification is available for only one item per channel; if you require more then you must open another connection. Notification is provided through the ON command; this command lets you pass control to another routine which can then use `DDEREQUEST` to obtain the actual data. For example:

```
ON KEY F1 GOTO DONE
DDEOPEN #1 "SERVER" "FILE1"
DDEADVISE #1 "Item1"
ON DDEADVISE #1 GOSUB GETITEM
ONLINE
:GETITEM
DDEREQUEST #1 "Item1" %1
MESSAGE ("Item1 is now " %1)
RETURN
:DONE
DDECLOSE #1
```

The DDECLOSE command

Syntax: `DDECLOSE <#dde>`

This command is used to close a DDE connection opened with the `DDEOPEN` command. The command needs just one parameter, the number of the DDE connection to be closed.

The DDEEXECUTE command

Syntax: `DDEEXECUTE <#dde> <string>`

This command is used to send a string over the specified DDE connection (which must have been created with the `DDEOPEN` command). The effect that the string to be executed (`<string>`) has will depend upon the DDE server to which you are connected. The `OK` variable will be set in the usual way to indicate success or failure of the command.

The DDENAME command

Syntax: `DDENAME <name>`

This command is used to change the application name used by the DDE interface. More information on how this is used is available in the *DDE Reference* appendix to the *User's Guide*.

The DDEOPEN command

Syntax: `DDEOPEN <#dde> <topic name>`

This command is used to connect to a DDE server, which may be either another instance of Glink, or a completely different application. As for any other DDE connection, you must specify an application and topic name. You must also specify a connection number, which will be used for all other commands using the DDE connection that is created by `DDEOPEN`. Nine simultaneous DDE connections are allowed, numbered from 1 to 9. The `OK` variable may be used to test whether or not a successful connection was made. If either the application or the topic name is left blank then the first matching server to respond will be connected to.

The DDEPOKE command

Syntax: `DDEPOKE <#dde> <item name> <item value>`

This command is used to send a data item to a DDE server (to which you first must have connected using the `DDEOPEN` command). You must specify both the name of the data item to be 'poked' and the value associated with the item.

The `OK` variable may be used to check whether the `DDEPOKE` command was performed successfully.

The DDEREQUEST command

Syntax: `DDEREQUEST <#dde> <item name> <%var>`

This command is used to request a data item from an open DDE connection (created with the `DDEOPEN` command). `<%var>` is the variable that is to accept the requested data. If the request fails for any reason, the `OK` variable will be set false.

The DEBUG command

Syntax: `DEBUG {ON|OFF|SEND xxx|RECEIVE xxx}`

This command turns the internal debugging (log of all characters sent and/or received) on or off (depending upon whether you use `DEBUG ON` or `DEBUG OFF`). Note that the `DEBUG` file is rewritten each time it is used, so you must save the results on another file (with `REN`, for example) before turning on the debug option for a second time. The debug information will be written to the file `GLINK.$DB` on the current directory (you may change this using the `SET DBGFILE` command), and is a raw dump of data in both directions. A change of direction on the communications line is marked in the debug file with a double vertical line character (hex BA) in the file. This character may be changed if you wish, using the commands:

```
DEBUG SEND "string"
DEBUG RECEIVE "string"
```

in which case the specified strings will be used to delimit the data in the debug file.

This file is the same file as is produced when you start Glink with the `/DEBUG` command line parameter, and `DEBUG OFF` may in fact be used to turn off debugging that was started from the command line. While debugging is in effect, Glink will display the word `'DEBUG'` at the beginning of the status line.

If you are using debug mode to document a problem that you are having with the emulator, remember that it's a binary file. It must remain that way if it's to be of any help to your support personnel, so be careful not to transfer the file in text mode.

The DEFAULT command

Syntax: `DEFAULT`

This command specifies the default action to be taken inside a `SWITCH` or `CSWITCH` construct when there are no matching `CASE` statements. It must be specified after all relevant `CASE` statements - see the `SWITCH` command on page 166 for an example.

The DEFINE command

Syntax: `DEFINE <number> <name>`

This allows you to specify an alternate name for a given variable. Variable names may be any length, but only the first eight characters are significant. Once `<number>` has been redefined in this way then `<name>` may be used instead of the number; this may be used to give names either to variables or to patterns. For example:

```
DEFINE 1 "Count"
...
ASSIGN %Count 1
IF (%Count gtn 10) GOTO DONE
...
```

Here we have given a name to variable `%1`, which may be referenced as `%Count` in this particular script. Note that this is only visible to this particular script - if it should call another script then the same `DEFINE` must be present if the same name is to be used. The actual variable is still `%1` and is still available as such, both in the called and in the calling script.

The DELAY command

Syntax: `DELAY <seconds>`

This command simply waits the specified number of seconds. Waiting for five seconds would look like this:

```
DELAY 5
```

If you need finer control of the delay time, you may use the `DTENTHS` command, or alternatively specify a time using a single position after the decimal point.

The DELMENU command

Syntax: `DELMENU <string>`

This command deletes any menu items that have been added to the main menu bar with the `ADDMENU` and `BUILDMENU` commands. The entry in the menu bar is deleted at the same time. This command is useful for the situation where the scripts defined in the menu bar are no longer relevant, or where a new set of scripts is needed, for example when logging into a new host. One parameter is required, the name of the entry you wish to delete (this either will be blank or will be the name you used when inserting the item with `BUILDMENU`). For example:

```
DELMENU ""           * delete 'user' menu bar item
DELMENU "&Mymenu"    * delete Mymenu item
```

The DFIND command

Syntax: `DFIND <string>`

This command may be used to find the contents of a dial directory entry given part of the host name (see `DREAD` for a way of reading the directory given the number of the desired entry). The directory will be searched starting at the first entry, and will return the first entry with a host name containing the string specified (the search is case-independent). For example, if you specified "tac" as the search string then "TAC", "Bull/TAC" and "The Tac System" would all be considered as matching entries.

If the command is unsuccessful then the `OK` variable will be false. If the entry is found, then the following internal variables will be set:

```
$COMMENT, $DIAL, $HOST, $LOGIN, $PASSWORD, $PHONE
```

The DIAL command

Syntax: `DIAL <number> [<attempts>]`

This command will dial the number configured at the corresponding position in your dial directory. Note that if there is an attached script to this number it will NOT be executed when the dialing is done from a script in this way. For example, to dial entry 5 in the directory you would use:

```
DIAL 5
```

`DIAL` will continue dialing until a connection is made, if no other parameters are supplied. You may restrict the number of times `DIAL` attempts to contact the host machine with an optional second parameter. For example:

```
DIAL 17 3
```

will make exactly three attempts to dial entry number 17, and give up if unsuccessful. You may test whether or not contact was actually established in this case using the `IF OK` statement.

The DISCONNECT (modem) command

Syntax: `DISCONNECT`

This command tells the emulator to disconnect the communications line. This is done by changing the status on the DTR line; if you are using a modem then make sure that it is configured to respond properly to this. Remember to enable the line again with `CONNECT` before the line is used again if the disconnection is not to be permanent. Note also that to disconnect from a normal network interface you should use the `NETDISCONNECT` command to disconnect from the host.

The DIVIDE command

Syntax: `DIVIDE <%var> <number>`

The `DIVIDE` command allows you to divide one number by another. The first parameter must be a script variable, while the second may be a script variable or a constant. The result of dividing the first number by the second is placed in the script variable specified first. For example:

```
DIVIDE %2 7
```

divides the present contents of the `%2` variable by 7, leaving the result in `%2`. Note that the result may be stored in exponential format to keep maximum precision. If you need to print a result that may be outside the range 0.01 to 32767, you can use the `TRUNCATE` command to format the number in a more suitable way. If the division can be performed correctly then the `OK` variable is set true. If not (because one of the two operands was non-numeric or because the second number was zero) then it's set false.

The DMARK command

Syntax: `DMARK {<number>|ALL|CURRENT}`

This command 'marks' the specified entry in the dial directory. In the usual way, this means that this entry will then be called when a queued dial is made, either by using the `RDIAL` script command or by using the queued dial facility in the dial directory manually. For example:

```
DMARK 3
```

marks the third entry in the dial directory. The special form `DMARK ALL` marks all entries in the dial directory with a single command, while the `DMARK CURRENT` form will mark the entry that you most recently have attempted to dial into (not necessarily last contacted).

The DOMENU command

Syntax: `DOMENU`

This command executes a menu that has been defined by the `MENU` and `MOP` commands (defined further on in this chapter). The menu is shown to the user, and the appropriate command is executed. After the command has been executed (unless it was a `GOTO` or `CHAIN` command) then control is returned to the statement after the `DOMENU` command. Control will also be passed to the statement after `DOMENU` if the user 'escapes' from the menu by pressing the Esc key, in which case the `$KEYPRESS` built-in variable will contain an escape character (`^[`). Under Windows the `$KEYPRESS` variable is also updated with enter (`^M`) or escape (^[) depending on when the user exits from the menu using the `OK` or the Cancel button. The `DOMENU` command has no parameters.

The DOS command

Syntax: `DOS <command>`

This will execute another Windows program (the name of the command is somewhat misleading but is necessary for compatibility with the DOS version of Glink -- you may prefer to use EXECUTE, which is a synonym for the DOS command).:

```
DOS "MYPROGRAM.EXE"
```

The program you start will be started in its own window, and your script will continue to execute in parallel with the Windows program you just started. If you need to synchronize your script with results provided only when the other window completes, the SET DOSWAIT command is provided to modify this behaviour. If you wish to control how the program you start should be displayed then the SET DOSSHOW command is available.

To use commands that are internal to COMMAND.COM under Windows you must invoke COMMAND.COM specifically. For example, to execute the delete command below you would use:

```
DOS "COMMAND.COM /C DEL MYFILE"
```

Also, commands that use the COM extension must be specified with the extension, for example to use the program LIST.COM the name must be specified explicitly.

For completeness, and explanation of how the DOS command works in the DOS version of GLINK is included here. For example, if you wanted to delete the file 'MYFILE' you would put the following into your script:

```
DOS "DEL MYFILE"
```

All normal DOS conventions like piping, etc., may be used. If you wanted to copy MYFILE to MYFILE.BAK but did not want this to write anything on your screen you could use:

```
DOS "COPY MYFILE MYFILE.BAK>NUL"
```

To perform operations like this with the Windows version of the program, you must be aware of the difference between internal and external commands. Internal Normally, DOS commands will be executed using the COMMAND shell; this is required in order to provide support for piping and for DOS internal commands, but has the disadvantage that not all error returns will be correctly reported. If you are dependent upon correct return values from DOS commands, then you should use the script command SET SPATH ON first; this tells GLINK to search the DOS PATH for the command itself (if the command is not found then the COMMAND shell will be called in any case). In this case you may test the status of the DOS command executed using the IF OK statement; also the DOS errorlevel reported by the program executed will be available in the $ERRORLEVEL built-in variable.

The DOSN command

Syntax: `DOSN <command>`

This command is kept for backward compatibility and is in all ways equivalent to the DOS command.

The DOWNLOAD command

Syntax: `DOWNLOAD <directory name>`

This command allows you to override the predefined download directory, normally to ensure that the files you collect finish up in the correct place. It is good practice to 'save' the current download directory before you start and restore it again when you are finished, thus:

```
ASSIGN %11 $DOWNLOAD
DOWNLOAD "C:\SENDHERE\"
....
DOWNLOAD %11
```

The DPATTERNS command

Syntax: `DPATTERNS`

This command resets all defined patterns (see `PATTERN`) to null values. These are preset to null when the script is started but it can often be useful to reset 'old' values before starting a new `MATCH` setup. Patterns that have been used in `WHEN` statements are **not** reset by this command.

The DREAD command

Syntax: `DREAD <number>`

This command may be used to read the contents of a dial directory entry given its number (see `DFIND` for a way of reading the directory given the host name). The number may be supplied either as a constant or as the name of a variable containing the number of the entry. If the command is unsuccessful then the `OK` variable will be false. If the entry is found, then the following internal variables will be set:

```
$COMMENT, $DIAL, $HOST, $LOGIN, $PASSWORD, $PHONE
```

The DSCREEN command

Syntax: `DSCREEN`

This command performs the 'dump screen' function that is available interactively on the `ALT+W` key. An optional parameter specifies the name of the file to which the dump should be directed. If it's absent, then the last file used for the screen dump function will be used. As usual, the current screen will be appended to the end of the file, preserving any previous contents.

The DTENTHS command

Syntax: `DTENTHS <tenths>`

This command simply delays for the specified interval. It is provided for those situations where you need finer control over the interval than is provided with the `DELAY` command. For example the command:

```
DTENTHS 15
```

would wait for one and a half seconds.

The DTIME command

Syntax: `DTIME <hour> <minute>`

This command stops execution of the script until the specified time. Two integers must be provided, the first specifying the hour and the second the minute. You must use a 24-hour clock when specifying the time. For example, to delay your script until nine o'clock in the evening you would specify:

```
DTIME 21 00
```

The command provides support for applications where unattended procedures need to be started automatically by the PC at a given time.

Note also the `SET WARNINGS OFF` command that should normally be used in unattended scripts.

The DUNMARK command

Syntax: `DUNMARK {<number>|ALL|CURRENT}`

This command removes the 'mark' for the specified entry in the dial directory. The specified entry will therefore not be called when the next queued dial is made, either by using the `RDIAL` script command or by using the queued dial facility in the dial directory manually. For example:

```
DUNMARK 3
```

removes the mark for the third entry in the dial directory. The special `DUNMARK ALL` form of the command removes marks for all entries in the dial directory with a single command. This will most often be used to clear marks preparatory to using the `DMARK` command to ensure that you have marked exactly the sites specified and no others, for example:

```
DUNMARK ALL
DMARK   1
DMARK   3
```

will result in exactly systems 1 and 3 being marked and no others, irrespective of the initial state of the dial directory. Another format, `DUNMARK CURRENT`, will unmark the most recently called system.

The DVARIABLES command

Syntax: `DVARIABLES`

This command resets all defined variables (not `PATTERNS` - see the `DPATTERNS` command on page 64) to null values.

The DWHENS command

Syntax: `DWHENS`

This command 'deactivates' any `WHEN` statements that might be active at the time.

The ECHO command

Syntax: `ECHO {ON|OFF}`

This command controls echoplex mode. When echoplex mode (`ECHO ON`) is in effect characters will not be echoed locally (in most cases they will be echoed remotely from the host machine). When echoplex mode is not in effect characters will be echoed locally as they are typed.

The EIGHTBIT command

Syntax: `EIGHTBIT {ON|OFF}`

This command may be used to change the state of the 'eightbit' host switch (see the description of this in the configuration documentation). It also may be used as a mechanism for loading both the 7-bit and 8-bit keyboard definitions. For example, if you want to run in 7-bit 'German' but load both the 7- and 8-bit keyboards (because you know that the host will switch you into 8-bit mode) then you could use the following:

```
EIGHTBIT ON
KEYBOARD DEF
EIGHTBIT OFF
KEYBOARD GER
```

The ELSE command

Syntax: `ELSE`

This is used inside a `BEGIN-ENDIF` block of statements to introduce the group of statements to be executed if the test in the `IF` statement fails. See the `IF` command on page 92 for an example of this.

The EMULATE command

Syntax: `EMULATE <string>`

The `EMULATE` command is in many ways similar to the `SHOW` command, with one minor technical difference. This is only interesting in situations where some of the characters in the specified string are using different codes on the PC and the host. Strings passed through the `SHOW` command are specified using the character set in use on the PC, while strings passed through the `EMULATE` command use the host character set. This can be useful for example when displaying the contents of a debug file.

The ENABLE command

Syntax: `ENABLE <keyname>`

A script that is waiting on an `ONLINE` command or because the pause key has been pressed, (see the `'PAUSE'` command) may normally be started off again by pressing ALT+O. If you wish to move this function to another key (for example because you would prefer to be able to start yet another script in this situation, or simply because you want to use a different key) then this is done with the `ENABLE` command. See the description of the `KEYS` command (page 100) for a list of valid key names. A special format, `ENABLE NONE`, will disable the 'script enabling key' altogether. Note that the `ENABLE` command has an effect not only for the current script but also for all subsequent scripts until another `ENABLE` command is executed.

The ENDBUILD command

Syntax: `ENDBUILD`

This command is used to mark the end of a group of statements introduced by the `BUILDMENU` command.

The ENDIF command

Syntax: `ENDIF`

This is used to mark the end of a group of statements that are to be executed following an `IF` statement. Normally only the first statement following the `IF` statement will be executed if the test succeeds, but use of a group of statements delimited with `BEGIN` and `ENDIF` overrides this. See the `IF` command on page 92 for an example.

The ENDSWITCH command

Syntax: `ENDSWITCH`

This is used to mark the end of a `SWITCH` or `CSWITCH` construct, and marks the place to which control will be given when a group of statements for a matching `CASE` have been executed (`SWITCH`) or when an `EXITSWITCH` command is executed (`SWITCH` or `CSWITCH`). See the documentation for the `SWITCH` commands (page 166) for more information.

The ENDWHILE command

Syntax: `ENDWHILE`

This is used to mark the end of a block of statements introduced with the `WHILE` command - see the documentation of that command for an example, page 173.

The ERASE command

Syntax: `ERASE <filename>`

The `ERASE` command allows you to erase a file (or files) without having to use the DOS command. `IF OK` may be used to test whether or not the command was successful. For example:

```
ERASE "TEMPFILE"
```

Wildcard characters ("?" and "*") may be used if you need to erase multiple files. For example:

```
ERASE "C:*.BAK"
```

The ERRORGOTO command

Syntax: `ERRORGOTO <label>`

This specifies the name of a label which is to be given control if a `RECEIVE` command should not find the string it is waiting for inside the timeout limit (see the `TIMEOUT` and `RECEIVE` commands). For example:

```
ERRORGOTO NOMATCH
```

If this command was in your script, any time a receive timed out you would continue processing at the label `NOMATCH`. Note that use of the `IDLE` or `ONLINE` commands will reset any active `ERRORGOTO`, which must then be reissued before the next receive for which it is to apply.

The EXECUTE command

Syntax: `EXECUTE <command>`

The `EXECUTE` command is a synonym for the DOS command.

The EXISTS command

Syntax: `EXISTS {FILE|DIR|ANY} <filename>`

The `EXISTS` command may be used to test for the presence of a named file or directory on your disk. If you specify `FILE` then the command will only check for the presence of files with a matching name; directories will not be included in the check. Similarly, if you specify `DIR`, then only directories, not files, will be included. If you specify `ANY` then both directories and files will be checked for. You may include the wildcard characters `'?'` and `'*'` in the file name in the command. The result of the check is returned in the `OK` variable.

For example:

```
EXISTS FILE "*.MSG"
IF OK GOTO SENDMSG
```

If any files (but not directories) with an extension of 'MSG' are found then the script will transfer control to the label SENDMSG.

The EXITSWITCH command

Syntax: EXITSWITCH

This is used to pass control from inside a SWITCH or CSWITCH construct to the corresponding ENDSWITCH command. Typically, inside a CSWITCH, the EXITSWITCH command (corresponding to 'break' in C) will be used at the end of each CASE group. It may also be used to exit immediately to ENDSWITCH in a group of statements inside a single CASE group. See the documentation for the SWITCH and CSWITCH commands for more information.

The EXTRACT command

Syntax: EXTRACT <#var> <parameters> <delimiters> <n>

This command scans the parameters provided and extracts parameter number <n> using the specified delimiters, placing the result in the given variable. More than one delimiter may be used as the same time. For example:

```
ASSIGN %1 "one,,three,four,,,seven"
EXTRACT %2 %1 "," 3
```

This would result in the value "three" being placed in variable %2.

The FCLOSE command

Syntax: FCLOSE <#file id>

This command closes one of the nine files that may be used inside a script. Only the file identifier need be specified. See the FOPEN command on page 75 for more information about file handling in Glink scripts. Example:

```
FCLOSE #1
```

The FCODE command

Syntax: FCODE <character>

This command sets the function code for the next message to be transmitted (this will only be relevant for synchronous interfaces). For example, if you need to send the exact equivalent of the HDS/BDS No key (which sends the escape sequence <esc>!, but also sets the function code to when used on a synchronous interface) then you would need to program this using:

```
FCODE "!"; TRNL "^[!"
```

The FILTER command

Syntax: FILTER <%var> <string>

This command allows you to remove selected characters from a string. The contents of the specified variable are inspected and any of the characters in the second string that are present in the first string will be removed. For example:

```
ASSIGN %1 "abcdefggfedcba"
FILTER %1 "ec"
```

will result in %1 containing "abdfggfdba". Similarly, you could remove all occurrences of the DEL character from the %2 variable using:

```
FILTER %1 "^$7F"
```

The FILTER command is always case sensitive, irrespective of the setting of SET CASE.

The FIND command

Syntax: `FIND <row> <column> <string>`

The `FIND` command searches for the specified string in the current screen, starting at the position indicated by the row and column parameters. The state of case checking (see `SET CASE`) is taken into account. A string that wraps across rows will ***not*** produce a match. If the string is found then its row and column coordinates will be returned in the built in `$FNDY` and `$FNDX` variables respectively. If the string is not found then both variables will be set to zero. An example of its usage is shown below, where we have a VT100 screen and wish to highlight all occurrences of a particular word.

```
Define 1 SaveX               * Variable definitions
Define 2 SaveY
Define 3 Name

Set Case ON                  * Case sensitive

Assign %SaveY $Y             * Save current position
Assign %SaveX $X
Assign %Name "Smith"         * String to search for

Find 1 1 %Name               * Find it
While ($FNDY gtn 0)          * While we have one
  Pos $FNDY $FNDX            * Go to display position
  Show ("^[[7m" %Name "^[[0m"]] * Show in inverse
  Find $FNDY $FNDX+1 %Name * Find next occurrence
EndWhile

Pos %SaveY %SaveX            * Reset screen position
Return
```

The FIX command

Syntax: `FIX <%var>`

The `FIX` command allows you to take a string (usually received directly from the line) and convert any embedded control characters into the `'^'` notation otherwise used for these in the script language. This can be useful when preparing a script in direct interaction with a host. For example, if the variable `%8` contained the value `'ABC'` followed by a carriage return and line feed, then the `FIX` command:

```
FIX %8
```

would change the contents of variable `%8` into `'ABC^M^J'`.

The FLOC command

Syntax: `FLOC <#file id> <%var>`

This command may be used when you are processing a file using the `FRDLINE` or `FWTLINE` commands. It returns the current line number in the file in the specified variable. If you have used any other commands than `FRDLINE` or `FWTLINE` on the file then the position is undefined. You may test whether a valid position was returned using the `IF OK` test. The line number returned is that of the record that is about to be read or written, in other words you will receive a value of one for a file that has just been opened.

The FLUSH command

Syntax: `FLUSH`

This command 'empties' any characters that may have been received from the line without displaying them on the screen.

The FNDEXEC command

Syntax: `FNDEXEC <%var> <filename>`

This command provides the full path to the executable that is associated with the file `filename`. For example if you specified `mydocument.doc` as the file name then the path to Microsoft Word would be placed in the specified variable.

The FNEXT command

Syntax: `FNEXT`

The `FNEXT` command (in combination with the `FSEARCH` command) can be used to extract a list of matching file names from a directory using wildcard specifications. `FNEXT` takes no parameters; it simply continues the search that was specified in the preceding `FSEARCH` command. If a new matching file is found, then the OK variable is set 'true' and the name of the file returned in the internal `$FILE` variable. Otherwise, the `OK` variable will be set 'false'. See the `FSEARCH` command on page 78 for an example.

The FOPEN command

Syntax: `FOPEN <#file id> <mode> <filename>`

Glink allows you to process up to nine files with your scripts. Files may either be read or written, and are associated with a file identifier with the `FOPEN` command. Files may be read with the `FRDBLOCK`, `FRDCHAR` and `FRDLINE` commands, and end of file testing done with the `IF EOF n` statement. Files may be written using the `FWTBLOCK` and `FWTLINE` statements. When you are finished with a file, it should be closed with the `FCLOSE` statement. `<#file id>` is a number from 1 to 9 that the other commands will use to identify the file. `<mode>` is one of `INPUT`, `OUTPUT`, `APPEND`, `IO` or `EXCLUSIVE` depending upon how the file is to be used.

<filename> is any valid filename. The EXCLUSIVE option specifies that although the file will only be used for input, exclusive access is needed in a networked or multitasking environment. If you need more specific sharing modes for FOPEN you can set these with the SET SHARE script command. The INPUT, IO and EXCLUSIVE open modes require that the file should already exist and will cause an error if it does not (use the EXIST command to check if necessary).

You may also use the conventional filename *CLP to access the clipboard contents as a file, where input operations will read data from the clipboard and output operations will set the clipboard contents.

You may also access communications ports using the FOPEN command. In this case the file name is specified as "*COMn:" where n is the number of the port. This will open the port with the current settings for that port, but you may also specify additional settings in the standard format used by the command shell MODE command:

```
COMn: [BAUD=b] [PARITY=p] [DATA=d] [STOP=s]
      [to=on|off] [xon=on|off] [odsr=on|off]
      [octs=on|off] [dtr=on|off|hs] [rts=on|off|hs|tg]
      [idsr=on|off]
```

For example you could use:

```
FOPEN #2 IO "*COM2: baud=19200 parity=N data=8"
```

If the FOPEN command fails then the OK variable will be set false, and in the special case of failure of FOPEN on the communications port an error message will also be available in the $DRESULT script variable.

As an example of how file commands may be used, the following small script takes the contents of the file "DUMP" and shows it on the screen, where "DUMP" may have any valid terminal commands embedded inside it:

```
FOPEN #1 INPUT "DUMP"      * Open the file for input
IF NOT OK GOTO ERROR       * Make sure it's OK
CLEAR                      * Clear the screen
WHILE NOT EOF #1           * Keep going until done
  FRDBLOCK #1 %1           * Read a piece of the file
  SHOW %1                  * Put it on the screen
ENDWHILE
FCLOSE #1                  * Close the file
```

The FPOS command

Syntax: `FPOS <#file id> <line number>`

This command may be used to position to a given line in a text file (the first line in the file is counted as line one). Note that the current line number in the file is available in the `$FPOS(#n)` built-in variable. For absolute positioning inside a file, see the `FSEEK` command on page 78.

The FRDBLOCK command

Syntax: `FRDBLOCK <#file id> <%var>`

This command reads up to 250 characters from the specified file and places them in a script variable.

The FRDCHAR command

Syntax: `FRDCHAR <#file id> <%var>`

This command reads a single character from the specified file and places it in a script variable.

The FRDLINE command

Syntax: `FRDLINE <#file id> <%var>`

This command reads a line of data (delimited by CR, LF, CR-LF, or LF-CR) from the specified file into a script variable. If the line contains more than 255 characters, it will be truncated.

The FSEARCH command

Syntax: `FSEARCH {FILE|DIR|ANY} <filespec>`

The `FSEARCH` command (in combination with the `FNEXT` command) can be used to extract a list of matching file names from a directory using wildcard specifications. The `<filespec>` parameter, which may use the normal '*' and '?' wildcard specifiers, specifies the name to match against. If a match is found, the `OK` variable will be set 'true', otherwise it will be set false. The search can be continued using the `FNEXT` command. In the case where a match is found, the internal `$FILE` variable will contain the name of the matching file. An example should make this clearer:

```
FSEARCH FILE "*.TXT"
WHILE OK
  MESSAGE $FILE
  FNEXT
ENDWHILE
```

This example simply lists all files in the current directory with an extension of `'TXT'`.

The FSEEK command

Syntax: `FSEEK <#file id> <location>`

This command may be used to move to an absolute location in a file (the first byte in the file is counted as byte zero). Note that the current location in the file is available in the `$FSEEK(#n)` built-in variable. For line-oriented positioning in text files, see the `FPOS` command on page 77. If you specify the location numerically you will be limited to a maximum position of 32767, for higher values use an intermediate variable, for example:

```
ASSIGN %1 262144
FSEEK #1 %1
```

The FSIZE command

Syntax: `<%var> <filename>`

This command returns the length of the named file (in bytes). If the file does not exist or for some other reason is not accessible, this command will return a length of zero bytes. For example, to check whether the file `MYFILE` in the current directory on the `C:` drive will fit into the floppy mounted in the `A:` drive before copying it, you could do the following:

```
FSIZE %1 "MYFILE"
CD "A:"
ASSIGN %2 $FREE
CD "C:"
IF (%1 GTN %2) GOTO ERROR
COPY "MYFILE" "A:"
```

The FSKIP command

Syntax: `FSKIP <#file id> <number>`

This command skips over `<number>` lines of data (delimited by any of CR, LF, CR-LF or LF-CR) on the specified file, and may be used for faster positioning inside the file.

The FTP command

Syntax: `FTP {COMMAND|EDIT|GET|OPEN|PUT|VIEW} filenames`

This command controls FTP-related procedures using the GlinkFTP client. You should set relevant options for the transfer using the `SET FTP` script verbs before executing the actual `FTP` command. "filenames" uses the same format as do the `GETFILE` and `PUTFILE` script commands, where in the case where both local and remote names are needed you ***must*** specify them both with a semicolon between, i.e. "local_name;remote_name". The various options available are:

`COMMAND` Executes the specified command on the remote FTP system. The syntax for the command is FTP server dependent.

`EDIT` Transfers the file to the PC for editing. When editing is complete the file will be transferred back to the host (if changed). The program used for editing is a configurable option in the GlinkFTP client.

`GET` Transfers the file from the host to the PC.

`OPEN` Transfers the file to the PC and executes the associated application (for example, if the file is a Word document then it will be opened in word).

`PUT` Transfers the file from the PC to the host.

`VIEW` Transfers the file from the host to the PC for viewing. The program used for viewing is a configurable option in the GlinkFTP client.

Note that by default the script will wait for the FTP procedure to complete before proceeding. You can change this using the `SET FTP NOWAIT` command. If you do this you can check on current progress using the `$FTPSTATUS` and `$FTPRESULT` built-in variables. If the transfer fails then an error message will also be available in the `$FTPERROR` variable.

```
 SET FTP ASCII
 SET FTP HOST "MYGCOS8"
 SET FTP NOWAIT
 FTP GET "TESTFILE.TXT;TESTFILE"
  ... other processing
 IF ($FTP EQN 0) GOTO CHECKFTP
  ... other processing
:CHECKFTP
 MESSAGE ("FTP finished with code " $FTPRESULT)
 IF ($FTPRESULT NEN 0) MESSAGE ("Err:" $FTPERROR)
```

The FVERSION command

Syntax: `FVERSION <%var> <filename>`

This command provides you with the software version number of the program residing in the file `filename`.

The FWTBLOCK command

Syntax: `FWTBLOCK <#file id> <string>`

This command allows you to write a block of data to the specified output file. No CR-LF delimiters are appended to the data; they are written exactly as they are stored in the script variable.

The FWTLINE command

Syntax: `FWTLINE <#file id> <string>`

This command is exactly the same as `FWTBLOCK`, except that a CR-LF delimiter is included at the end of the data.

The GETDATE command

Syntax: `GETDATE <%var> <format string>`

The `GETDATE` command allows you to use today's date in a variety of ways. The `<format string>` tells Glink in exactly which format you would like the date to be delivered. Inside the format string, you use any of the following:

`YYYY`	year
`YY`	last two figures of year
`MMM`	month in text format
`MM`	month as two digits
`M`	month as one or two digits
`DD`	day as two digits
`D`	day as one or two digits
`WWW`	day of week, text
`OOO`	ordinal day of year
`J`	Julian date

Any other characters in the format string will be reproduced as they appear there. Some examples of format strings and the resulting output for March 13th, 1989:

`YYMMDD`	890313
`WWW D. MMM, YYYY`	Mon 13. Mar, 1989

`D/M-YY`	13/3-89

The GETENV command

Syntax: `GETENV <%var> <string>`

This command may be used to collect the value of a environment variable (if defined). For example, to collect the name of the currently specified command shell into variable `%5` you could use:

```
GETENV %5 "COMSPEC"
```

The GETFILE command

Syntax: `GETFILE <protocol> <filename>`

This command starts a file transfer from the host to the PC. Two parameters are required, first the type of transfer to start, and secondly, the name of the file to transfer. Note that in the case of FTP transfers you should use the `FTP` script command.

Valid keywords for the file transfer protocol are:

`ASCII`	ASCII capture
`DPROT`	Use default protocol
`FTRA`	FTRA (GCOS7/GCOS8 Kermit) transfer
`GMODEM`	Ymodem-G transfer
`IND$FILE`	IBM 3270 IND$FILE transfer
`KERMIT`	Kermit transfer (see `BINARY ON/OFF`)
`KSERVE`	Kermit transfer when other side is server
`MOD7`	Modem7 transfer
`TELINK`	Telink transfer
`XMODEM`	Xmodem transfer
`YMODEM`	Ymodem transfer
`YBATCH`	Ymodem-batch transfer
`ZMODEM`	Zmodem transfer

For example, to download the file `'MYFILE.ARC'` using Xmodem you would use the command:

```
GETFILE XMODEM "MYFILE.ARC"
```

For the Kermit protocol only, the file name may be specified in the format `"PC-name;Host-name"`, in which case the name used for the host machine may be different from the name used locally on the PC.

Remember also that when starting a file transfer from a script you must ensure that the host side of the transfer has been initiated before executing the `GETFILE` or `PUTFILE` script command. For example, to transfer a file `'TESTFILE'` on a GCOS8 host to `'TESTFILE.TXT'` on your PC, you might code:

```
TRNLINE "FTRAN PC7800"
GETFILE FTRAN "TESTFILE.TXT;TESTFILE"
```

in your script. The situation for IND$FILE transfers is somewhat more complex in that you must ensure that the command you send 'comes from' the correct field in the form. A fairly consistent way of doing this is to simulate entry of cursor home, cursor back tab, clear field, entry of the command, and transmit. Remembering that the host file name is already specified in the command sent to the host so that you only need to specify the PC file name in the `GETFILE` command, this could result in (for example):

```
SHOW "^[H^[[Z^[K"
SHOW "IND$FILE GET AAX5PRD.ACCNT(J9307) ASCII CRLF"
SHOW "^i"
GETFILE IND$FILE "J9307.TXT"
```

In the above, `"^[H"` is the home function, `"^[[Z"` is the back tab function, `"^[K"` is the field clear function, and `"^i"` is the transmit function. The actual format of the IND$FILE command you will need to send will vary from host to host; a complete description of these will be found in the chapter describing file transfers.

The GETKEY command

Syntax: `GETKEY <string>`

This command waits for one of a specified set of keys to be pressed before continuing. Only keys in the supplied list will be accepted at this point, and the actual key pressed will be available for the script in the built-in $KEYPRESS variable. The supplied parameter may be either a variable or a constant string containing any alphanumeric characters. The comparison with the key pressed is case-independent.

For example:

```
GETKEY "ABC"
SWITCH $KEYPRESS
  CASE "A"; MESSAGE "A was pressed"
  CASE "B"; MESSAGE "B was pressed"
  CASE "C"; MESSAGE "C was pressed"
ENDSWITCH
```

The GETLENGTH command

Syntax: GETLENGTH <%var> <string>

This command allows you to collect the length of a variable into another script variable. <string> is the entity whose length is to be collected (may be a string, variable, built-in variable, etc.).

The GETMACRO command

Syntax: GETMACRO <%var> <N>

This command allows you to collect the value of the specified macro number N into a script variable.

The GETTIME command

Syntax: GETTIME <%var> <format string>

In the same way as the GETDATE command lets you collect today's date, the GETTIME command allows you to place the time of day into a script variable. <format string> tells Glink how you would like the time to be formatted.

The following strings are interpreted:

`C`	tick counter (18.2 ticks per second, since midnight)
`HH`	hours as two digits (24-hour clock)
`H`	hours as one or two digits (24-hour clock)
`UU`	hours as two digits (12-hour clock)
`U`	hours as one or two digits (12-hour clock)
`MM`	minutes as two digits
`M`	minutes as one or two digits
`SS`	seconds as two digits
`S`	seconds as one or two digits
`FF`	hundredths (Fractions) of seconds as two digits
`F`	hundredths (Fractions) of seconds as one or two digits
`AP`	replaced by 'am' or 'pm' depending on time

Some examples of possible input formats and their results for a time of 12:05 in the afternoon:

`HHMMSS`	120500
`U.MM AP`	12.05 pm
`HH:MM`	12:05
`C`	791980

Two additional modifiers may be supplied in the formatting string. The first of these, the exclamation mark (!), is used to tell Glink that the specified variable already contains a time, which MUST have been collected as a 'tick counter' using the 'c' format. The output in this case will not be the current time of day, but the elapsed time since the original counter was collected. The format for this second `GETTIME` may be any of the available formats, however. For example:

```
GETTIME %1 "c"                * starts a timer
  *(perform some script commands)
GETTIME %1 "!mm:ss.ff"        * collects elapsed time
MESSAGE ("Elapsed time = " %1)
```

The second available modifier is the percent sign (%), which tells Glink that the call timer (elapsed time since last login) is to be used rather than the time of day. For example:

```
DISCONNECT
GETTIME %1 "%hh:mm"
MESSAGE ("Time spent online = " %1)
```

The GETVALUE command

Syntax: `GETVALUE <%var> <string>`

The `GETVALUE` command provides you with a way of converting binary data into an integer string. Often this will be needed because you've read binary data direct from a file and need to have it available in integer format.

If `<string>` is less than four characters long, then the entire contents will be converted into a number. If it contains four or more then only the first 4 characters will be converted. The conversion assumes that the least significant byte occurs first in the string.

For example, assume that you need to use the value corresponding with hexadecimal 0xFE43 in your script. Although there is no direct provision for entering hexadecimal numbers, you could use the following:

```
ASSIGN %1 "^$43^$FE"
GETVALUE %1 %1
```

Note the order in which the actual contents of the string were entered here.

The GETWORD command

Syntax: `GETWORD [MAIN | SCROLLBACK] <row> <column>`

The `GETWORD` command provides you with an easy way of collecting a word from the screen or scrollback buffer. The row and column coordinates entered may specify any position in the word, and all relevant data about the word found will be returned as follows:

`$WDL`	Length of word
`$WDX1`	Column for first character of word
`$WDX2`	Column for last character of word
`$WDY1`	Row for first character of word
`$WDY2`	Row for last character of word
`$WORD`	Contents of actual word

By default, words are considered to be delimited by spaces, but you can specify additional characters that should be considered as delimiters by setting them in the screen options configuration or by using the `SET DELIMITERS` script command. Wrapping will be done between screen lines (hence the need for returning row values both for the start and end of the word). If the specified screen position contains a delimiter then only that single delimiter will be returned. As for other script commands, you may read the status line by using a zero as the row number for the main screen, and lines in the scrollback buffer are numbered from zero for the most recent line and upwards. Negative numbers may also be specified as parameters when reading the scrollback and these will be interpreted as referring to the equivalent area of the main screen (in which case the returned parameters in `$WDY1` and `$WDY2` will also be negative).

For example, if you wish to collect the word at the current cursor position, mark it, and copy it to the clipboard, you could do:

```
GETWORD MAIN $Y $X
SET MARKMODE NEW
SET MRECT OFF
MARK $WDX1 $WDY1 $WDX2 $WDY2
PERFORM COPY
```

Another example - let's assume that you want to provide a function that adds a new entry to the right hand mouse button menu using CONTEXT. The new function is to collect that word without marking it and put it back into the current screen at the current cursor position. You can do all of this by executing the following command (all on one line):

```
CONTEXT MAIN ADD "=GETWORD MAIN $CY $CX; SHOW
$WORD" "Collect Word"
```

The GOSUB command

Syntax: `GOSUB <label>`

The `GOSUB` command behaves like the Basic command of the same name. Processing will continue at the specified label rather than continue inline, but Glink will remember the position of the `GOSUB` command. The label may be anywhere within the script file, either before or after the `GOSUB` command (but it may not be in another script file). When the subroutine that was invoked with `GOSUB` has finished whatever it is doing, it performs a `RETURN` and processing continues at the statement after the `GOSUB`. For example:

```
 GOSUB R1
 SEND "2"
 ...
:R1
 SEND "1"
 RETURN
```

would send first a "1" and then a "2".

The GOTO command

Syntax: GOTO <label>

The GOTO command just transfers control to the label defined with the same name. This may be anywhere in the file, before or after the GOTO. Note that you can't GOTO a label in another script file.

The GPARAM command

Syntax: GPARAM <string>

The GPARAM command sets the value of the global parameter. This parameter exists even while scripts are not running and can therefore be used to save information between two different scripts that will execute at different times. It may also be used to set information externally that will apply to all scripts that may run subsequently. The value set using GPARAM can be retrieved from the internal variable $GPARAM (which returns the entire string) or using one of the $Gn parameters which treat the value as a sequence of fields with space separators. For example, if one script set the global parameter using:

```
GPARAM "F1 F2 F3 F4"
```

a script executing later would be able to refer to $GPARAM (in which case the value "F1 F2 F3 F4" would be returned) or to $G3 (which would return the value "F3").

The GPROFILE command

Syntax: `GPROFILE <%var> <filename> <section> <item>`

This command allows you to read options from INI files that are formatted in the same way as for example the `SYSTEM.INI` file, with sections introduced using headings in the format

```
[section-name]
```

The command returns the value of the selected option in the `<%var>` variable. For example, if the `[boot.description]` section of your `SYSTEM.INI` file contains the following line:

```
codepage=850
```

then executing the following script command:

```
GPROFILE %1 "C:\WINDOWS\SYSTEM.INI"
   "boot.description" "codepage"
```

would place `"850"` in script variable number 1. If the specified item does not exist then the contents of the script variable will remain unchanged, so normally this should be initialized with the default value of the option you are collecting.

The GWCONNECT command

Syntax: `GWCONNECT <ip-address> <hostname>`

The `GWCONNECT` command is used in connection with the gatewayed interfaces, and will connect to the specified host. Its primary use is in connection with Ggate gateways. In this case it takes two parameters, first the symbolic or numeric IP address of the host on which the gateway is running, and second the defined name of the host which the gateway should provide a connection to.

For example, if you have a host defined as `DPS8` and your gateway software is running on the machine with IP address 192.150.211.7, then you would use:

```
GWCONNECT "192.150.211.7" "DPS8"
```

If you already are configured with the correct gateway IP address then you may specify this with a null string, and the current value will be used. For example:

```
GWCONNECT "" "DPS7"
```

Note that if you wish to set the IP address or host name without actually connecting to the host then the `SET SERVER` and `SET RESOURCE` commands may be used for this purpose. Also, if you wish to use the defined profile for a particular host but override one or more of the host parameters then you may do this by including the relevant parameters after the host name. For example:

```
GWCONNECT "" "DPS8 -DN PH02"
```

If you are using one of the DNTD gateways, you may optionally specify the name of the gateway PC as the first parameter, and specify the installation ID for the DPS6000 machine as the second. For example:

```
GWCONNECT "" "DPS6-1"
```

would connect to the machine named DPS6-1 both on the NetBIOS and the SPX gateways using the default gateway machine, while

```
GWCONNECT "192.150.211.35" "DPS6-2"
```

would do the same for the TCP/IP variant of the DNTD gateway (in this case the IP address of the gateway machine is obligatory).

The `GWCONNECT` command can also be used to connect to a TNVIP gateway. In this case, the first parameter is the host address, and the second address is the TNVIP resource name:

```
GWCONNECT "192.150.211.35" "ISP4"
```

In this case, you may also set the terminal type required for the host connection using the `SET TTYPE` command.

The HALT command

Syntax: `HALT`

This command simply stops the emulator. If the configuration has been changed then the emulator will not ask for confirmation that the new configuration should be saved. If you need to save changes made by your script then use the `CONFIG SAVE` script command first. No operator intervention should be required when the `HALT` command is used. If a termination script (`$$TERM.SCR`) is present it will be executed (unless the command is present in the termination script itself).

The HOST command

Syntax: `HOST <string>`

This command allows you to insert your own text in the space normally reserved for the host name in the status line. The text will only appear if the configuration option for show host name has been selected. For example:

```
HOST "Login complete"
```

For special purposes you may use a URL in the host name field. If you do this then a message will be displayed when the mouse passes over the field and the field will be clickable (when clicked the URL will be opened using the defult browser).

The ICON command

Syntax: `ICON <icon file> <icon number>`

This command lets you define the default icon that Glink will display when minimized. `<icon file>` is the name of the file containing the icon, and `<icon number>` is the number of the icon inside the file. If you have no tools that let you investigate the contents of icon files then trial and error will also work. For example, to get a generic comms icon from the `MORICONS.DLL` file delivered with Windows you can use:

```
ICON "C:\WINDOWS\SYSTEM32\MORICONS.DLL" 67
```

If you wish to load an icon from an icon file (with an extension of `.ICO`) then use zero as the icon number in that the file contains only a single icon.

The IDLE command

Syntax: `IDLE <seconds>`

This command tells the emulator that it should wait until the host has been 'idle' for the specified number of seconds before continuing ('idle' in the sense that the host no longer sends any data). The time may be specified with a resolution of 0.1 seconds. The normal use for this would be in a situation where the host is sending a stream of data, but is not providing any constant string at the end of the data stream that allows you to find out when the transmission is finished. For example:

```
SNDLINE "STATUS"          * ask for some host data
GETF ASCII "LOGFILE"      * turn on logging
IDLE 2.5                  * wait until 2.5 secs idle
CAPTURE OFF               * turn off logging
```

Note that the `IDLE` command will reset the target of any active `ERRORGOTO` or `ON TIMEOUT` statements, which must therefore be reissued before any new `RECEIVE` statements are used if the target is still to be active.

The IF command

Syntax: `IF [NOT] <condition> <script command>`

This is used for various testing purposes. In all of them the same rule applies - if the test is true then the following command will be executed; otherwise it will not. The first use of the `IF` statement is to check whether a match was found on one of the patterns you have defined with a `PATTERN` command. If the pattern was matched then the command in the `IF` statement will be executed; otherwise processing will continue from the statement in the following line. For example:

```
PATTERN !1 "You have mail"
RECEIVE "RDY:"
IF !1 GOTO CHECKMAIL
```

In this case, if the string "You have mail" is received at some point prior to the "RDY:" text then control will be passed to the CHECKMAIL label. The syntax 'IF NOT !n' is also valid and has the meaning you would expect.

The next form of the IF command may be used to check on the status of a preceding file transfer (GETFILE/PUTFILE command) or dial (DIAL/RDIAL) command:

```
IF [NOT] OK <command>
```

This form may also be used after a CMPNUM to check whether both variables had numeric values, and after a SCAN command to check whether or not the second string was contained in the first.

A related test is:

```
IF FAIL <command>
```

This test is a specific test provided for those using the RDIAL command and is explained there.

Testing for a keypress from the user may also be done with:

```
IF KEYPRESS <command>
```

Note that the IF KEYPRESS statement does not actually 'read' the key involved, so you may use the test several times in succession, to skip out of a set of embedded subroutines for example. The key will still be 'pressed' and the script will react in whatever way it would otherwise have done. To check for a keypress and at the same time 'dispose' of the key, use WKEY as soon as possible after the IF KEYPRESS test.

Another format is available in connection with the file handling commands, and may be used to test whether EOF has been reached on an input file. The syntax for this one is:

```
IF [NOT] EOF <#file id> <command>
```

where <#file id> is the file identifier for the file in question (see the FOPEN command for more information).

Yet another format is used to check upon the results of CMPNUM and COMPARE operations:

```
IF [NOT] {EQ|NE|LE|LT|GE|GT} <command>
```

Here only one of the comparison tests is used, where the symbols have the following meanings:

`EQ`	The variables are equal
`NE`	The variables are not equal
`LE`	The first variable is less than or equal to the second
`LT`	The first variable is less than the second
`GE`	The first variable is greater than or equal to the second
`GT`	The first variable is greater than the second

Such comparisons need not be coded as separate `COMPARE` and `IF` statements; you may include the two strings or numbers to be compared in parentheses directly after the `IF` command. For example:

```
IF (%1 EQ "A") GOTO FOUND
```

To distinguish between normal comparisons and numeric comparisons (`COMPARE`s and `CMPNUM`s) an extra set of operators is provided:

```
EQN, NEN, LEN, LTN, GEN, GTN
```

which have the same meanings as the equivalent operators without the additional `'N'`; these force a numeric compare, while the normal set of operators will provide a normal string comparison. Also, note that the use of one of these constructs in an IF statement will leave the result of a comparison available for further testing in the same way as `COMPARE` and `CMPNUM`. This means that something like:

```
IF (%1 EQN 10) GOTO EQUALS
IF LTN GOTO LESSTHAN
```

is legal. The comparison need not be done again, in other words.

If you are using the 'direct' comms interface, and a modem which is set up to reflect the 'true' status of the carrier signal, then you may also use the following format of the `IF` command to check whether you are still logged into the host:

```
IF OFFLINE GOTO LOGGEDOFF
```

The IF OFFLINE test may also be used on LAN interfaces, in which case it will test whether or not you actually are connected to a host machine. An equivalent IF ONLINE statement is also available.

If you have been waiting for a string to arrive from the host (for example with a RECEIVE command) then you may test whether or not a timeout occurred using:

```
IF TIMEOUT GOTO ERROR
```

The TIMEOUT status is set true whenever a timeout occurs, and set false whenever a new statement that might incur a timeout is executed.

The script language provides a SET STATUS command (see the SET command) allowing you to set an internal flag. This flag may be tested later in a script using IF TRUE and/or IF FALSE.

Normally an IF statement will execute the single command following the IF statement, independently of how things are formatted on the line. A common error is therefore to write something like:

```
IF (%1 EQ %2) ASSIGN %5 "Y"; GOTO P1      * WRONG!
```

In this case, the GOTO will be performed irrespective of the result of the test comparing %1 and %2. The only script statement affected by the comparison is the ASSIGN statement immediately following the IF. In the case where you need to do this kind of thing, the extended IF syntax may be used:

```
IF [NOT] <condition>
BEGIN
  (statements)
ELSE
  (statements)]
ENDIF
```

This allows you to execute a group of statements as the result of a single test, and optionally, to execute another group of statements if the test fails. For example:

```
IF ($KEYPRESS EQ "1")
BEGIN
  ASSIGN %1 "One"
  MOPTION 1
ELSE
  ASSIGN %1 "Invalid"
  MOPTION 2
```

```
ENDIF
```

Further IF statements may be nested inside such groups of statements up to a maximum of 50 levels.

The INCLUDE command

Syntax: `INCLUDE <filename>`

The INCLUDE command allows you to collect source from another file that will be inserted into the script in place of the command. This can be useful when you for example have a list of common DEFINE statements that you wish to share between a suite of scripts. Note the difference between INCLUDE and CALL: whereas the INCLUDE command copies in the source code directly and thus produces a single script, the CALL command starts a totally different free-standing script when the first script is executed. The INCLUDE command should be followed by the name of the script file to be included, for example:

```
INCLUDE "\SCRIPTS\DEFINES.SCR"
```

The code copied from the included file is processed exactly as though it had replaced the INCLUDE file. You may nest INCLUDE statements to any depth, limited only by available memory.

The INFILE command

Syntax: `INFILE <%var> <string> [<max length>]`

The INFILE command is exactly the same as the INPUT command except for one thing. In the case of INFILE the user will be allowed to press F10 in order to extract a file name from the file display. This is done in the same way as for file name entry other places in the emulator (in the Windows versions of the emulator an additional button will be added to the dialog box for the same purpose). For example:

```
INFILE %3 "File to send: "
```

For more details, see the INPUT command below.

The INPC command

Syntax: `INPC <%var> <string> [<max length>]`

This command is exactly the same as the INPUT command, with the exception that the cursor is placed at the END of any predefined input in the editing window. This can be used for values that you do not expect the user to have to override, or where it's natural for the user to add to the predefined input rather than overwrite it.

The INPUT command

Syntax: `INPUT <%var> <string> [<max length>]`

This command collects input from the keyboard and assigns it to the specified variable. You must specify both the number of the variable to which the input is to be assigned and a 'prompt' which should tell you what kind of input is expected. For example, to collect the name of a file later to be downloaded you could use the command:

```
INPUT %1 "Please enter your name: "
```

to assign the user's input to variable number 1. You may precede the INPUT command with an MPOS command to control the position of the input window on the screen. An optional additional parameter may be used to specify the maximum length of the input to be read, for example:

```
INPUT %2 "Name (max 6 characters): " 6
```

You may preinitialize the value to be used in the window simply by assigning a value to it before using the INPUT routine. In the same way, if you do **not** want anything to appear as a default then you should ASSIGN a value of "" before calling the INPUT routine. The cursor will be placed at the start of any such input value, so that if the user starts typing immediately then the old input will disappear (but see the INPC command).

If the user 'aborts' the input routine (by pressing Esc, or in Windows by pressing the Cancel button) then a null string is returned in the specified variable. The $KEYPRESS built-in variable will also contain either a CR (^M) or escape (^[) character depending on the way the user exited. This still applies under Windows, even if the user actually exited using the mouse.

The title shown on the top line of the menu window may be set using the MENU command. If no title is required, and you have been using menus, precede the INPUT command with a MENU "", as you will otherwise 'inherit' the title of the previous menu.

The INPUT commands may be used inside a menu definition, in which case the syntax is slightly different - the prompt field is then not specified in that the MOP or MOPC command being used will already have provided this. In this case, the maximum length for input is a required parameter. For example, you could specify:

```
MOP "File name: " INFILE %1 32
```

in which case space will be set aside inside the menu for the 32 character input field. Additionally, in that INFILE was used in this example, the F10 key to list the current directory (and possibly select a file name) will be available while positioned inside this field.

The INVISIBLE command

Syntax: INVISIBLE <%var> <string> [<max length>]

This command is the same as the INPUT command in all respects but one; the characters typed by the user do not appear on the screen but instead appear as asterisks or question marks. This is especially useful for collecting passwords. For example:

```
INVISIBLE %3 "Please enter password: " 12
```

The ISOCONNECT command

Syntax: `ISOCONNECT <connect parameters>`

This command is specific for the Nixdorf/NFS and NCR/OSI communications interfaces, and will start the procedure required to connect to the specified host. The command accepts one parameter, which consists of a number of fields separated by commas (,) or semicolons (;). The parameters are specified in the following order:

- Front End
- Local TSAP
- Remote TSAP
- Local SSAP
- Remote SSAP
- Terminal ID
- Password

The user will be presented with the normal connect menu, filled out with those parameters that were specified in the `ISOCONNECT` command. The cursor will be positioned in the first blank field. For example, using

```
ISOCONNECT "DN100,TM01,,X1,TSS"
```

will present a menu with the fields for Front End, Local TSAP, Local SSAP and remote SSAP filled out, with the cursor placed in the Remote TSAP field.

The ISSERVICE command

Syntax: `ISSERVICE`

If you have set Glink up to run as a service, for example to run background scripts independently of a user being logged on, then you should include the ISSERVICE command at the start of your scripts. This will tell Glink to display any message boxes that may be produced in a way that is compatible with services and which otherwise would not be displayed on the screen.

The KEYBOARD command

Syntax: `KEYBOARD <keyfile>`

This command allows you to change to another keyboard definition. The keyboard is loaded into the 7-bit or 8-bit keyboard definition depending on the mode that currently is set. The '.glinkxlit' extension should not be included; for example to load the PC character set you would use:

```
KEYBOARD "KPC"
```

You can use the OK variable to test whether loading the keyboard definition was successful.

The KEYKERMIT command

Syntax: `KEYKERMIT <keyfileid>`

This command sets the keyboard translation file to be used for Kermit text transfers. Note that if you use any of the options that specify transfers for the OEM character set in the Windows versions then that translation will be performed in addition to the translation specified here. If no translation is specified for Kermit transfers, either with this command or with the setup option, then the current 8-bit keyboard will be used.

The KEYS command

Syntax: `KEYS <keyname> [<keyname> [...]]`

The `KEYS` command allows you to simulate the action of any keys on the keyboard while in a script. The `KEYS` command takes a variable number of parameters, one for each key you need to emulate, with spaces between each keystroke. Normal keys use just their keytop names, and the prefixes `ALT-`, `ALTGR-`, `SHIFT-` and `CTRL-` may be used to specify the relevant shift keys (for example `SHIFT-F3` or `ALT-M`). These prefixes may in turn be prefixed with `L` or `R` if you wish to refer to the left or right version of the key specifically where this is relevant, for example `LALT-Q`. Note that if an asterisk (*) is used in this command, then it should be enclosed in quotes or it will be treated as the start of a comment.

Glink recognizes in addition the following names:

BS	backspace
CENT	center keypad on enhanced keyboard
CR	carriage return
DEL	the DEL key
DOWN	down arrow
END	the END key
ENT	ENTER
ESC	the ESC key
F1-F12	function keys, combine with ALT/SHIFT/CTRL as needed
GR*	Num * on numeric keypad
GR+	Num + on numeric keypad
GR-	Num - on numeric keypad
GR/	Num / on numeric keypad
HOME	the Home key
INS	the Ins key
LEFT	left arrow
LF	Line feed
MLDBL	double click on left mouse button (see note)
MLEFT	left mouse button
MLUP	left mouse button up
MMDBL	double click on middle mouse button (see note)
MMIDDLE	middle mouse button down
MMOVE	mouse move
MMUP	middle mouse button up
MRDBL	double click on right mouse button (see note)
MRIGHT	right mouse button
MRUP	right mouse button up
MWHEEL	mouse wheel rotation, rotation value returned in $MWHEEL
PGDN	the PGDN key
PGUP	the PGUP key
PRT	the PRINT SCREEN key
RIGHT	right arrow
SP	space
TAB	the TAB key
UP	up arrow

In the case where a key has two separate versions on the keyboard, for example the Page Down key which can be found both on the extended function pad and on the numeric keypad, then the KEYS command will refer to the basic version of the key (in this case the version on the numeric keypad). If you have a specific need for the extended version of the key then prefix the standard name with the letter X. For example:

```
KEYS XPGUP
```

Keys for which this applies are DEL, DOWN, END, ENT, HOME, INS, LEFT, PGDN, PGUP, RIGHT and UP.

Note that the 'mouse' keys above are designed for use with the ON script command and are included in this table for the sake of completeness. Note that when the mouse is trapped using one of these commands this disables normal mouse usage. The user can however still perform normal mouse actions by holding down the CTRL key at the same time as using the mouse. Note also that you may only trap double clicks if you have also trapped the equivalent single click.

For example, if you wanted to duplicate the actions of a user tabbing to the third field of a form, entering 123, and pressing transmit, you could use the following script command:

```
KEYS Home Tab Tab 1 2 3 Gr+
```

Note the spaces between each of the letters and numbers in these commands - each argument to the KEYS command must be a 'keypress'; strings are not accepted here.

Note that the KEYS command may only be used to pass data to the emulator window, not to dialog boxes or other Windows programs.

Beware of using the KEYS command to enter simulated keys as a result of an ON KEY command, especially if the typeahead queue has been enabled. The keystrokes may well not be entered in the order you expect. If you can use the SHOW command to perform the *result* of the keystroke by passing an escape sequence to the emulator you will usually find that to be more reliable.

The LABEL command

Syntax: `LABEL <label>`

This defines a place in the script that you can get to using a `GOSUB` or a `GOTO` command. Names of labels may be any length, but Glink will only check the first 8 characters. You may not have more than 250 labels in a single script file. Labels may be specified either by using `LABEL` written in 'longhand' or by using a colon (:) in the first non-blank position, in other words:

```
LABEL GOHERE
:GOHERE
```

mean just the same thing.

The LAYOUT command

Syntax: `LAYOUT <layout name>`

This command allows you to switch to another keyboard definition. The name you specify is used wth the extension '.glinklayout' to specify the name of the keyboard layout file you are using (for example, `LAYOUT "L01"` would tell Glink to load the keyboard layout from `L01.glinklayout`). A `LAYOUT` command with no parameters tells Glink to load the default keyboard layout (`DEF.glinklayout` if it exists; otherwise the default layout provided with the emulator).

You can use the OK variable to test whether loading the keyboard layout was successful.

The LCASE command

Syntax: `LCASE <%var>`

This command can be used to convert a variable into lower case (small letters). It just needs one parameter, the variable to be converted, for example:

```
LCASE %1
```

Note that for the conversion of 'high ASCII' characters to be performed correctly then you must have defined your country code and codepage correctly, or some characters may not be interpreted.

The LINE command

Syntax: `LINE <string>`

The `LINE` command takes the string provided and provides it to the emulator exactly as if the string had been received from the host machine.

The LOCAL command

Syntax: `LOCAL {ON|OFF}`

The `LOCAL ON/OFF` command turns local mode on and off. Note that while `LOCAL` mode is effective, all data from the line will be ignored.

The LOG command

Syntax: `LOG {ON|OFF|TOGGLE}`

This command controls print logging mode. If print logging is enabled then data coming from the line will be sent in parallel to the configured printer (control sequences will be stripped).

The MANDIAL command

Syntax: `MANDIAL <string>`

This command allows you to dial a number using the dial directory functionality but avoiding the necessity of including an entry in the dial directory with the appropriate number in that the number is specified in the command. For example:

```
MANDIAL "22419764"
```

would call the number 22419764. If you would like to use the alternate number functionality of the dial directory a second number may be supplied separated from the first with a '/' character. In the same way, if an additional modem command is needed this may be specified after a second '/' character. For example, the command

```
MANDIAL "22410403//ATS95=2"
```

would dial 22410403 after sending the additional modem command ATS95=2.

All the normal dial directory functions apply, in particular dial codes and abbreviations.

The MARK command

Syntax: `MARK X1 Y1 X2 Y2`

This command is used to set up a mark in the current screen. `X1` and `Y1` are the column and row coordinates of the start of the mark, while `X2` and `Y2` are the column and row coordinates of the end of the mark. The type of mark depends on whether rectangular marking has been set or not (either using the Edit menu or using the `SET MRECT` command).

In older versions of the program, the calculation of the mark position was not consistent with the values returned by the `$MXn` and `$MYn` built-in variables. In practice you therefore had to use a value for `X2` one greater than you would expect (also one greater for `Y2` if rectangular marking was set). This is still the case, for reasons of compatibility, but if you would prefer to be able to use consistent values then we have provided the `SET MARKMODE` command. For example:

```
SET MARKMODE NEW
MARK 1 1 3 1
```

will set a mark covering the first three columns of line one, irrespective of whether rectangular marking is being used.

The MATCH command

Syntax: `MATCH`

This is used after you have set up some patterns with the `PATTERN` command. `MATCH` tells Glink to watch what the host is sending, and to wait until one of the specified patterns turns up. At this point processing will continue. You can use the `IF` command to find out which of the patterns was found. For example:

```
PATTERN !1 "CONNECT"
PATTERN !2 "BUSY"
PATTERN !3 "NO CARRIER"
PATTERN !4 "VOICE"
SNDLINE "ATDT123456"
MATCH
IF !1 GOTO CONNECTED
etc...
```

The `MATCH` command is by default case sensitive, but this may be changed using `SET CASE OFF`.

If a `SET IDLE` command is active then Glink will wait for the line to be idle for the specified time before continuing to execute the script (this may be found necessary on half-duplex lines).

The MBAR command

Syntax: `MBAR <menu> <item id> {ENABLE|DISABLE|LOCK}`

The `MBAR` command allows you to disable (and enable) specific options in the Glink for Windows menu bars. `<menu>` in the above specifies which menu you are referring to and is one of:

`MAIN`	The menu bar in the main emulator window
`SCROLLBACK`	The menu bar in the scrollback window
`FILE`	The menu bar in the file display
`DIAL`	The menu bar in the dial directory
`SYSTEM`	The system menu

The identifier <item id> specifies the item using a positional notation (this is to avoid problems with national language versions of the software). Each entry in the top level of the menu is numbered starting from 1. The same applies to popup menus attached to these, and in this case a '/' separator is used to indicate the extra level. In popup menus with horizontal separation bars, the bar should be counted as an extra position. For example, to disable the 'Settings' option (the third entry in the main menu bar) you would use:

```
MBAR MAIN 3 DISABLE
```

On the other hand, if you only wished to disable the selection of a different communications interface (the second option in the second item of the 'Settings' menu item) then you would use:

```
MBAR MAIN 3/2/2 DISABLE
```

The 'LOCK' option provides an additional level of security - once an option has been locked then subsequent attempts to enable the option will fail. For example if you wished to disable the two items for script procedures and script commands you would use:

```
MBAR MAIN 1/10 LOCK
MBAR MAIN 1/11 LOCK
```

Note that two horizontal separator bars have been counted in these commands. If you are disabling items in the main 'Help' menu item then additional menu items that may have been added with the BUILDMENU command will not be counted, in other words the Help item is always item 6. If you wish to access such additional menu items you may do so, with item numbers of 7 and up; for example, the second additional menu item would be accessed with an item number of 8.

The MCURSOR command

Syntax: MCURSOR <cursor name>

This command allows you to select any of the standard Windows cursor shapes for the main emulation window. The following are supported for <cursor name>:

ARROW	Standard arrow cursor

`CROSS`	Crosshair cursor
`HAND`	Hand cursor
`IBEAM`	Text I-beam cursor (default)
`ICON`	Empty icon
`NESW`	Double-pointed cursor with arrows pointing northeast and southwest
`NS`	Double-pointed cursor with arrows pointing north and south
`NWSE`	Double-pointed cursor with arrows pointing northwest and southeast
`SIZE`	Square with a smaller square inside its lower-right corner
`UPARROW`	Vertical arrow cursor
`WAIT`	Hourglass cursor
`WE`	Double-pointed cursor with arrows pointing west and east

The MD command

Syntax: `MD <directory name>`

The `MD` command allows you to create a new working directory without having to use the `DOS` command. `IF OK` may be used to test whether or not the directory was created successfully. For example:

```
MD "\TEMP"
```

The MDIAL command

Syntax: `MDIAL <string>`

This command lets you set (or remove) the modem dial command. It uses one parameter, the dial command you wish to set. For example:

```
MDIAL "ATDT"
```

This command can be especially useful when setting up a configuration that must function both on network interfaces (where you would often wish to remove the dial command in order to allow the dial directory to function as a selection mechanism for network addresses) and also directly to a modem using a different dial directory. Removing the dial command can be done quite simply with the command

```
MDIAL ""
```

The MENU command

Syntax: `MENU <title>`

The MENU command defines the start of a user-defined menu, which may be used to present the user with a set of alternative actions. A user-defined menu must always start with a MENU command, followed by a number of MOP or MOPC commands (defining the different options to choose between), or MTEXT commands which just present text, and finally by a DOMENU command, that actually presents the user with the resulting menus. Optionally, the MPOS command may be used to predefine where on the screen the menu is to appear. If you are using the Windows versions of Glink, you may also use the MFONT command to select the font in which the menu will be displayed.

A typical sequence for defining a menu could be:

```
MPOS 3 3
MENU "Please choose one of:"
MOPC "A" "Log into system A"   GOTO P1
MOPC "B" "Log into system B"   GOTO P2
MOPC "C" "Log into system C"   GOTO P3
DOMENU
```

The menu will be shown to the user at the time the DOMENU command is executed. At this point the user may position on one of the options with the arrow keys and press ENTER to select that option; the associated command will be executed and control will pass to the statement after DOMENU. Alternatively, if ESC is pressed, control will pass directly to the statement after DOMENU without executing any of the commands defined with MOP and/or MOPC.

Menus are always left on the screen as long as possible. This will in general be until you start using the actual emulation functions again (SHOW, MESSAGE) or receive data from the host (RECEIVE, MATCH, RCVLINE, etc). This allows you to 'tile' menus on top of each other and also to leave a menu on the screen while some lengthy process is being performed by the script. Up to 8 menu windows may be displayed simultaneously, with the limitation that the total area occupied by these eight menus must not exceed two full screens of data. Extra control of such tiling for the expert user is provided with the UNMENU, REMENU, and NOMENU commands.

The MENU command is also used to set titles for input windows generated by the INPUT, INFILE, INPC, and INVISIBLE commands.

The MESSAGE command

Syntax: `MESSAGE <string>`

The MESSAGE command will display the string specified on the terminal, followed by a CRLF sequence. The string will not be sent to the host.

The MFONT command

Syntax: `MFONT <fontname> <pointsize>`

The MFONT command defines a font to use when displaying script menus. Note that current versions of Windows do not allow you to choose italic or light fonts, in that the internal procedures select an upright bold font if one is available. For example, if you choose Arial Italic and Arial Bold is available then Arial Bold will be displayed. The selection of a specific font is not only useful for aesthetic reasons, but may also be useful when you are preparing a menu where you need to align items exactly and can use a non-proportional font to do so.

Note that font selected remains active for all subsequent menus, not only in the script selected, but also in scripts executed later in the same Glink session. If your script does not want to affect the execution of other scripts then it should reset the font when finished with it. Some examples:

```
MFONT "Courier New" 12
MFONT "Arial" 17
MFONT "" 0              * selects the default font
```

The MINIT command

Syntax: `MINIT <string>`

This command allows you to set the modem initialization string. It's provided mainly for usage by the first-time user script, but may be useful in a situation where more than one modem is being addressed by the same copy of the emulator. Note that the initialization string is only sent to the modem when the emulator starts up, unless the option in modem setup for reinitialization before each outgoing call is set. For example:

```
MINIT "AT&F&C1&D2S0=0"
```

The MODE command

Syntax: `MODE <emulation mode>`

This command switches the emulator between the different emulation modes. Valid forms of the identifiers for `<emulation mode>` are:

```
VIP
V77
7107
7102
3270
5250
3151
ANSI
VT220
PRESTEL (or VIEWDATA)
MINITEL (or TELETEL)
```

The MOK command

Syntax: `MOK`

When you define a menu containing only `MTEXT` commands, the normal action for a script is to continue execution (leaving the menu displayed where this is possible). Often however you simply want to display a text of some kind and want the execution of the script to be delayed until the user has confirmed that this should happen. The `MOK` command is designed to provide this functionality. An 'OK' button will be added to the menu, and further execution delayed until the button is pressed.

The MONO command

Syntax: `MONO {ON|OFF}`

The `MONO` command acts in the same way as the command line `/M` switch. In other words, it tells the program to use only black and white on the screen (whether or not there apparently is a color screen attached).

The MOP command

Syntax: `MOP <string> <script command>`

The `MOP` command defines an option in a user menu. It should always be preceded by a `MENU` command and followed later in the script by a `DOMENU` command, which actually starts execution of the menu. `<string>` is the text to present in the menu, and `<script command>` is the command to execute if the option is selected by the user. The maximum number of different options you may select 22. Given that the maximum length of a single text is 76 characters, this enables you to provide a menu that may be anything up to a complete screen (not including the status line).

The MOPC command

Syntax: `MOPC <character> <string> <script command>`

The `MOPC` command defines an option in a user menu. In addition to providing exactly the same functions as the `MOP` command, you define an additional alpha-numeric character. This character may be used to choose this option directly from the menu, rather than have to position the cursor to the option first. If the defined character occurs in the menu text (as a capital letter) then that letter will be presented in high-intensity. See the `MENU` command on page 109 for an example.

Normally, if a key is pressed that is not defined in a `MOPC` command for the current menu, then no action will be taken. If you wish to provide a 'default' exit in the menu to cover other possible keys typed by the user, then the special format:

```
MOPC "" "string" <command>
```

may be used. The command specified here will be executed whenever a character not specified in one of the other MOPC statements for the current menu is typed (the actual character typed will be available in the `$KEYPRESS` built-in variable).

The MOPTION command

Syntax: `MOPTION <position>`

Normally the first option in a script menu will be where the cursor is placed when the menu is presented to the user. `MOPTION` allows you to override this and place the cursor on one of the other available options. Used together with the `$MOPTION` internal variable this may also be used as a 'place saver' for a menu that has been removed from the screen but should be presented again at a later time.

The MOVEWINDOW command

Syntax: `MOVEWINDOW <X> <Y> <DX> <DY>`

The `MOVEWINDOW` command allows you to move and/or resize the emulator window. `X` and `Y` are the new coordinates for the top left hand corner of the window, while `DX` and `DY` are the new width and height of the window. All values are in pixels. Note that the current values for all of these are available in the internal variables `$WX`, `$WY`, `$WDX` and `$WDY`. If you wish to move the window without resizing you can therefore use a command like this:

```
MOVEWINDOW %NewX %NewY $WDX $WDY
```

Resizing the window without moving would be done with much the same type of command:

```
MOVEWINDOW $WX $WY %NewDX %NewDY
```

The MPOS command

Syntax: `MPOS <row> <column>`

Normally menus are presented centered on the screen, but you may change this using the `MPOS` command. If the menu cannot be presented in the position you require then it will be presented centered instead. This applies both vertically and horizontally, and this fact may be used if what you in fact intend to do is present a menu on a given row but centered horizontally, for example (in which case you can just specify 79 as the column position).

The `MPOS` command may also be used to control the position of the `INPUT`, `INFILE`, `INPC` or `INVISIBLE` user input windows. It must always be used immediately before the command that presents the menu or input box (for example, you must repeat the `MPOS` if you use a `REMENU` command).

The MSGBOX command

Syntax: `MSGBOX {CAPTION|HIDE|POS|SHOW|TEXT} ...`

The `MSGBOX` command allows you to display a small message box, typically to keep the user informed of work in progress. The box will remain on display until it's either hidden using the `MSGBOX HIDE` command or until the script terminates. It will be centered on the screen by default but this may be overridden using the `MSGBOX POS X Y` command, where X and Y are specified as pixel coordinates. Normally you will set the caption and text for the message box, show it on-screen, possibly change the text once or twice and then finally hide it when you are finished:

```
MSGBOX Caption "TP8 TEST"
MSGBOX Text "Connecting to host..."
MSGBOX Show
NETCONNECT
MSGBOX Text "Connected, setting up the session..."
ON TIMEOUT 10 GOTO FAILED
SET IDLE 0.2
SNDLINE ""
CONVERSE "$$ 4800 MODEL: " "VIP7804"
CONVERSE "$%$" "CN TP"
MSGBOX Text "Logging into TP..."
CONVERSE "LOGICAL ID--" "AA01"
CONVERSE "***HELLO***" "LOGON"
MSGBOX Hide
```

The MTEXT command

Syntax: `MTEXT <string>`

Lines on menus that have corresponding actions are presented with the `MOP` and `MOPC` commands. Lines that should just contain text but are not options that may be selected to perform a particular action may be presented with the `MTEXT` command. The line occupies a space on the menu but will not be 'selectable' with the arrow keys. You may in fact choose to build a menu that contains no `MOP` or `MOPC` instructions at all, in which case the menu will be put onto the screen, but the script will continue executing immediately. This may be used to present a text 'in a box' in a simple and easy way (note also that you can use `MOK` to make the script wait at this point if this is your intention). For example:

```
MENU " File Error "
  MTEXT "Could not access the specified file!"
  MTEXT "Press enter to continue"
  MOK
DOMENU
```

The MULTIPLY command

Syntax: `MULTIPLY <%var> <number>`

The `MULTIPLY` command allows you to compute the product of two numbers. The first parameter must be a script variable, while the second may be a script variable or a constant. The result of multiplying the two numbers is placed in the script variable specified first. For example:

```
MULTIPLY %11 %12
```

multiplies the present contents of the `%11` variable by the present contents of the `%12` variable, leaving the result in `%11`. Note that the result may be stored in exponential format to keep maximum precision. If you need to print a result that may be outside the range 0.01 to 32767, you can use the `TRUNCATE` command to format the number in a more suitable way. If the multiplication can be performed correctly then the `OK` variable is set true. If not (because one of the two operands was non-numeric) then it's set false.

The MVSCROLL command

Syntax: `MVSCROLL <X> <Y> <DX> <DY>`

The `MVSCROLL` command allows you to move and/or resize the scrollback window. `X` and `Y` are the new coordinates for the top left hand corner of the window, while `DX` and `DY` are the new width and height of the window. All values are in pixels. Note that the current values for all of these are available in the internal variables `$SWX`, `$SWY`, `$SWDX` and `$SWDY`. If you wish to move the window without resizing you can therefore use a command like this:

```
MVSCROLL %NewX %NewY $SWDX $SWDY
```

Resizing the window without moving would be done with much the same type of command:

```
MVSCROLL $SWX $SWY %NewDX %NewDY
```

Note that the actual size of the window may be adjusted slightly to accommodate a whole number of columns and/or rows of screen data.

The NAME command

Syntax: `NAME <string>`

This command allows you to insert a name in the status line in the space normally used for the user name (see the `/N` startup option). Example:

```
NAME "Joe"
```

For special purposes you may use a URL in the user name field. If you do this then a message will be displayed when the mouse passes over the field and the field will be clickable (when clicked the URL will be opened using the defult browser).

The NETCONNECT command

Syntax: `NETCONNECT  <host name>`

This command is provided for those using network interfaces, and allows you to make a connect using a specified server name. It uses one parameter, the 'server name' to connect to. For example, on the Atlantis X.25 interface, you could establish a connection to the X.25 address 031069 using the following command:

```
NETCONNECT "031069"
```

The `NETCONNECT` command may also be used with a null parameter in the case where you simply want to reconnect to the server to which you were most recently connected. You may check whether the connection was successful using the `IF ONLINE` script command.

The NETDISCONNECT command

Syntax: `NETDISCONNECT`

This command is provided for those using network interfaces, and disconnects you from the host into which you are currently logged.

The NEW command

Syntax: `NEW <scriptname>[!label]`

This command also allows you to give control to another script file in the same way as the CHAIN command, but unlike the CHAIN the new script will execute at top level, so that a RETURN command will just terminate rather than give control to an upper level script. For example:

```
NEW "SCRIPT4.SCR"
```

The whole pathname must be provided (except the SCRGL extension, which is optional). Note that all of the normal conventions for starting a script apply. A special use of the NEW command is to ensure that the script is executing at top level (and will not see interference from code in a calling script):

```
IF ($LEVEL gtn 1) NEW $SCRIPT
```

The NOMENU command

Syntax: `NOMENU`

The `NOMENU` command removes all active menus from the screen immediately. Normally when you are using menus, the system will keep the current menu on the screen until it must be removed (see explanation under `'MENU'`). If you wish to remove it immediately, either because you wish to execute another menu or remove the menu(s) as a signal to the user, then the `NOMENU` command will do this for you.

The OBJECT command

Syntax: `OBJECT <filename>`

Including an `OBJECT` command in a script tells Glink that the script is not to be executed, but just compiled, and saved on the specified file. This file can then be used as a script in just the same way as a normal script. There are several reasons for wanting to do this; firstly, the generated file is smaller and is saved 'ready to go'. In the case of large scripts this can save valuable time. Secondly, the compile process 'garbles' the output (including any text that may be included there). This fact can be used to either make sure that the script is not modified by another user, or to protect the contents of the file from inspection to discover login sequences and the like.

Note that scripts saved in compiled format will usually have to be recompiled for new releases of the Glink software. This is because the format used to save the compiled script will not necessarily be compatible between one release and the next. A warning message will be issued if an attempt is made to execute an incompatible compiled script.

The OEM command

Syntax: `OEM`

This command specifies that the current script is using the native PC character set rather than the Windows (ANSI) character set. Scripts using 'high-ASCII' characters that were prepared using a non-Windows editor should include this command (which takes no parameters) to be compatible with the Windows editors.

The OLE command

Syntax:

```
OLE Create ObjectName "Class.Name"
OLE Connect ObjectName "Class.Name"
OLE Get [<%var>|ObjectName] ObjectName.PropertyName
OLE Set ObjectName.PropertyName <value>
OLE Call ObjectName.MethodName [parameters]
OLE Calr [<%var>|ObjectName] ObjectName.MethodName
[parameters]
OLE Free ObjectName
OLE ERRORS {NORMAL | FATAL | NONFATAL}
```

This command provides the functionality you need if you wish to use OLE automation from your script. To use the above commands you will first have to establish a connection with the OLE object you wish to control, using either the `CREATE` or `CONNECT` command. In both cases, you will specify a name that you choose yourself for the object, which you will use in the other `OLE` commands. If the command fails then the script `OK` variable will be set false and a descriptive error text will be made available in the `$OLERESULT` variable.

While controlling the object you may use `GET` to access properties that are exposed by the automation server, `SET` to modify them, and `CALL` or `CALR` to use its methods (use `CALR` when the method returns a value that you need). If the call actually returns another object then you should specify a symbolic name rather than the name of a script variable.

When you specify parameters for the methods you are using, you may prefix them with `N=name` to specify a named parameter, and also `T=x` to specify that you wish the following parameter to be delivered as an integer (this is required for some of the earlier OLE automation servers). The type 'x' can be S for a string, I for an integer or B for a boolean. In general Glink will inspect the values being delivered and attempt to deliver them in an appropriate format unless you format them specifically with the 'T=' syntax.

You may free the object when you are done with it using `OLE FREE`; the object will in any case be freed when the script terminates.

You can control rutime error handling using the `OLE ERRORS` command. Normal is to terminate the script with an error message except for `CREATE` and `CONNECT` which will set the `OK` variable and `$OLEERROR/$OLERESULT` variables. `FATAL` tells Glink to terminate on any `OLE` error, `NONFATAL` to not terminate, setting `OK`, `$OLEERROR` and `OLERESULT`.

The subject of OLE in general is too wide to be dealt with in any detail here, and the specifics are obviously specific to the actual automation server you are accessing, but we can take a look at the sort of things you can do by providing some examples:

This example opens Word, reads in a document and copies the entire document to the clipboard. It then starts a file transfer to a GCOS8 host to transfer the contents to a text file `doc1` on the host machine:

```
Ole Create Word "Word.Application"
Ole Call Word.Documents.Open "C:\MYDOCS\DOC1.DOC"
Ole Call Word.ActiveDocument.Range.Copy
Ole Call Word.ActiveDocument.Close 0
Ole Call Word.Quit
Ole Free Word
Binary OFF
Set Transfer ANSI
SndLine "GKRM F"
PutFile FTRAN "*CLP;doc1"
```

This next example assumes that we are logged into a host that is displaying some tabular data (the contents of the table will of course affect the statements we use to format the table). The data is marked, copied to the clipboard, pasted into an instance of Word, converted to a table with some simple formatting applied, and saved as `table.doc`:

```
Set Mrect ON
Set Ctabs ON
Mark 8 5 72 22
Perform Copy
Ole Create W "Word.Application"
Ole Set  W.Visible 1
Ole Call W.Documents.Add
Ole Call W.ActiveDocument.Range.Paste
Ole Call W.ActiveDocument.Range.Select
Ole Call W.Selection.ConvertToTable ^I
Ole Call W.Selection.HomeKey 6
Ole Call W.Selection.Cells.Delete 3
```

```
Ole Call W.Selection.SelectColumn
Ole Set  W.Selection.ParagraphFormat.Alignment 2
Ole Call W.Selection.MoveRight 12 4
Ole Call W.Selection.SelectColumn
Ole Set  W.Selection.ParagraphFormat.Alignment 2
Ole Call W.ActiveDocument.SaveAs "TABLE.DOC"
Ole Call W.ActiveDocument.Close 0
Ole Call W.Quit
Ole Free W
```

You will find a working example of the OLE script command in the Scripts\Demo\Ole.Sub example in your Glink installation.

The ON command

Syntax: See below

The ON command has two main uses; firstly as a way of enabling specific actions that are to be taken for certain keys while the script is running, and secondly to give more advanced possibilities than ERRORGOTO when handling timeouts.

To enable a key, use the ON command like this:

```
ON KEY <keyname> <another command>
```

You will find a list of valid keynames documented under the KEYS script command. <another command> can be any valid script command. You could for example write:

```
ON KEY F1 GOSUB RF1
```

and let the RF1 subroutine do whatever processing you wish. Note that the keys that are 'reprogrammed' by ON KEY are only 'active' when the script itself is waiting for input; i.e. you are executing a RECEIVE, MATCH, ONLINE or similar command. Up to 40 different keys may be defined with ON KEY. Mouse button presses as well as keypresses may be intercepted in this way, for example:

```
ON KEY MLEFT GOSUB MENU1
```

specifies that whenever the user presses and releases the left-hand mouse button, the MENU1 subroutine will be executed. To react to ANY keypress from the user, you can use:

```
ON KEYPRESS <another command>
```

Note that when you use this syntax the `ON` event is 'reset' and that you should reissue the `ON KEYPRESS` to trap subsequent keypresses.

Beware of using the `KEYS` command to enter simulated keys as a result of an `ON KEY` command, especially if the typeahead queue has been enabled. The keystrokes may well not be entered in the order you expect. If you can use the `SHOW` command to perform the *result* of the keystroke by passing an escape sequence to the emulator you will usually find that more reliable.

The `ON` command also provides more advanced possibilities than the `ERRORGOTO` command for handling timeouts, as noted above. The simplest usage of this type is:

```
ON TIMEOUT <seconds> <another command>
```

`<Another command>` can be any valid script command, and n is the maximum number of seconds the script should wait. The equivalent function to an `ERRORGOTO` would thus be `ON TIMEOUT n GOTO ...` However, if you have other requirements for error handling you can use this statement to provide them. For example:

```
ON TIMEOUT 10 SNDLINE ""
```

would send a CR message if the required string were not found within 10 seconds. If you have an active `ON TIMEOUT` command you may use the `TIMEOUT` command to change the actual timeout period without changing the associated action. Note that use of the `IDLE` or `ONLINE` commands will reset any active `ON TIMEOUT` command, which therefore must be reissued if it is to remain active after use of either of these commands.

The next usage for the `ON` command is to test for 'activity':

```
ON INACTIVITY 300 GOTO LOGOUT
```

This says that if there has been no activity of any kind, either from the line or from the keyboard, within the specified time, then the specified statement should be executed.

Another use:

```
ON SESSION 3600 GOTO WARNING
```

Each time you log into a new machine, a 'session' timer is started. The specified statement is executed when the timer expires.

To check for a disconnection, the `ON` command looks like this:

```
ON OFFLINE GOTO LOGGEDOFF
```

On most comms interfaces, the ON ONLINE and ON OFFLINE statement means that the rest of the ON statement will be executed if the program should detect a connection or disconnection with the host. On direct or modem connections, these are triggered when the carrier signal changes. The example below shows how a script could automatically reconnect when it gets disconnected:

```
:N1 ON ONLINE GOTO N2; NETCONNECT; ONLINE
:N2 ON OFFLINE GOTO N1; ONLINE
```

Incoming DDE advises can be checked with the following `ON` command:

```
ON DDEADVISE #1 GOSUB GETITEM
```

The rest of the `ON` statement will be executed when a DDE Advise message is received from the DDE server program.

The ON command may also be used for prints:

```
ON ENDPRINT GOSUB WORDPRINT
```

This command will be executed whenever a local or host print terminates. This command is useful when printing to a file. When the ENDPRINT is activated, the print file is closed, making it accessible to your script for copying or printing via other applications using OLE, DDE or a simple command line such as NOTEPAD.EXE /P “myfile.prt”. The print may be terminated by the host, by a local print screen, by the print timer or by turning the print logging off.

You can detect receiving the ‘turn’ from the host by using:

ON TURN GOSUB CHECKSCREEN

When this command is used, the script will call the subroutine whenever the turn is received from the host. The detection of the turn depends on the communications interface being used and the emulation mode. If the emulation mode is FORMS or TEXT or the communications interface supports the notion of TURN (e.g. Ggate, TNVIP, TN3270, TN5250), then it will be immediate, otherwise the turn will be triggered 500 milliseconds after the host has stopped sending data. In both cases, the turn will only be generated if the keyboard is unlocked.

Active ON events may be reset at any time using the script RESET command. For example, an active ON OFFLINE statement can be cancelled using RESET OFFLINE.

The ONLINE command

Syntax: `ONLINE`

The `ONLINE` command allows Glink to continue in interactive terminal mode yet still retain control in the script procedure. Any `WHEN` statements that are active at the time the `ONLINE` command is executed will still be active and perform their defined functions. This allows you to 'monitor' the line for predefined sequences even though the emulator is still being used for normal purposes. The script will continue executing at the statement after the `ONLINE` command when the user presses the 'enable' key. The enable key is predefined as `ALT+O` but this may be changed with the `ENABLE` command. An example of the use of `ONLINE`:

```
PATTERN !1 "^M^JNO CARRIER"
WHEN !1 GOTO LOGOUT
ONLINE
```

Note that the `ONLINE` command will reset the target of any active `ERRORGOTO` and `ON TIMEOUT` commands, which therefore must be reissued when appropriate.

The PACE command

Syntax: `PACE <milliseconds>`

This command will set the 'line pacing' (in milliseconds). This can be useful if you are connecting to systems where the speed at which you can send data varies in different parts of the system. For example:

```
PACE 10
```

The PARAM command

Syntax: `PARAM <string>`

This command sets the value of the 'script parameter'. Normally you will specify parameters to scripts simply by including these in the `CALL` or in the command line you use to start the script. In some cases you may wish to modify the parameters used after the script has started, or in the case of the UVTI interface simply use the parameter as an alternative way of delivering data to the external procedure.

The PARITY command

Syntax: `PARITY {EVEN|ODD|NONE|EVN8|ODD8}`

This command will set the parity on the communications line. The following options are supported:

`EVEN`	7-bits even parity
`ODD`	7-bits odd parity
`ONE`	8-bits no parity
`EVN8`	8-bits even parity
`ODD8`	8-bits odd parity

You only need to use this if you need to change the parity setting from the setting that was active when the script was started.

The PATTERN command

Syntax: `PATTERN <!pattern> [<string>]`

Up to twenty different 'patterns' may be set using this command, numbered from 1 to 20. Glink will look for any patterns that may be set any time you do a MATCH (which says wait until a pattern is received) or a RECEIVE (which specifies what to wait for, but Glink will still be looking out for the patterns you have set). For example:

```
PATTERN !3 "You have mail"
```

To reset a pattern:

```
PATTERN !3
```

You need to be able to reset patterns because Glink will always look for any pattern that has a defined value when you use the MATCH command. Note that DPATTERNS will reset ALL your patterns (except for those with active WHEN statements - to delete all patterns absolutely use DWHENS before using DPATTERNS). Another main use of patterns is in connection with the WHEN statement. WHENs are used to set up 'actions' that will be performed when the pattern is recognized even though the script is performing some other action at the time. Patterns are case sensitive by default, but this may be changed using the SET CASE OFF command. Note that this applies when the pattern is actually set, not at the time the comparison is actually made.

The PAUSE command

Syntax: `PAUSE <keyname>`

The PAUSE command allows you to specify a key which may be used interactively to 'pause' a script (see the KEYS command for a list of valid keynames). Normally there is no 'pause' key defined. However, as soon as one script has defined a 'pause' key, that key will remain in effect for subsequent scripts that may be executed. This will be the case until the key is redefined with another PAUSE command or disabled with a PAUSE NONE command.

Note that if the script was waiting for a certain text at the point where it was 'paused', re-enabling the script with the ALT+O key (or the key defined with the ENABLE command) will resume execution at the command AFTER the wait. In other words, the wait will be 'broken'.

The PERFORM command

Syntax: PERFORM {functionname}

The PERFORM command allows you to execute any of the internal functions of the emulator. These functions are exactly the same as those internal functions that can be assigned to the keyboard using the keyboard configurator. Many of them are also available through other script commands, but the complete set is supplied here for completeness. Note that the PERFORM command is more reliable than the KEYS command for these functions in that they are not subject to keyboard redefinition. They are specified as follows:

ASCIITABLE	ASCII table
BREAK	Send break
C132	132 cols swap
CALLTIME	Call timer
CAPTURE	Capture data toggle
CHAT	Chat mode toggle
CLEAR	Local screen clear
COPY	Windows copy function
DEFMACRO	Inline macro
DIAL	Dial a number
DUMPSCREEN	Screen dump
EDTBACK	Edit mode back
EDTFORWARD	Edit mode forward
EDTMODE	Enter edit mode
EDTRECALL	Edit command recall buffer
EXEMACRO	Exec inline macro
FILES	Show files
HANGUP	Hang up line
HELP	Help menu
INFO	Program info menu
INSTOGGLE	Insert toggle
KILLALL	Kills previous sessions (*Ggate* only)
KSERVE	Kermit server
LINEBUFF	Show line buffer

`LDIALOG`	Local dialog menu
`LOCTOGGLE`	Toggle local mode
`LOGIN`	Send login name
`MAC-0` to `MAC-63`	Keyboard macro
`MHELP`	Top level help menu
`PASSWORD`	Send password
`PASTE`	Windows paste function
`PRINTSCREEN`	Print screen
`PRTVAR`	Print variables only
`PSEND`	Windows paste/send function
`PXMT`	Windows paste/transmit function
`RECEIVE`	Receive a file
`RESET`	Reset error
`RESTART`	Restart comms
`RULER`	Toggle crosshair ruler
`SAVESCROLL`	Save in scrollback
`SCRDOWN`	Scroll down
`SCRLEFT`	Scroll left
`SCROLLBACK`	Display scrollback
`SCRRIGHT`	Scroll right
`SCRUP`	Scroll up
`SEND`	Send a file
`SETUP`	Setup menu
`STARTSCRIPT`	Start a script
`SWITCH`	Switch session
`TERMINATE`	Terminate emulator
`TOGGLES`	Toggles menu
`TRANSFER`	File transfer menu

The KILLALL function may require extra coding to wait for the Ggate host to accept and disconnect the host sessions before you re-initiate a connection, e.g.:

```
SET SERVER "ggate.gar.no"
PERFORM KILLALL
:W1
  IDLE 0.1
  IF ONLINE GOTO W1
GWCONNECT "" "MYHOST"
```

The PICK command

Syntax: `PICK {FILE|DIR|ANY}`

This command may be used when you wish the user to select a file using the file display menu. The file display is immediately called and the name the user selects with the ENTER key will be returned in the `$FILE` built-in variable. You may specify whether a file or directory should be selected (or either), and the header shown in the file display will reflect which choices are valid.

If no file is selected then a blank value will be returned. For example:

```
PICK FILE
IF ($FILE eq "") GOTO ALLDONE
PUTFILE ASCII $FILE
```

The file or directory must exist in order to be `PICK`ed.

The PLAY command

Syntax: `PLAY <filename>`

This command plays the named waveform file or system sound. For example, to play the waveform file `TADA.WAV` supplied with Windows you would use:

```
PLAY "C:\WINDOWS\MEDIA\TADA.WAV"
```

To play the 'Exit Windows' sound you would use:

```
PLAY "SystemExit"
```

If the specified sound does not exist then it will be replaced with a normal beep.

The POPUP command

Syntax: `POPUP`

This command removes the most recent entry from the call stack. What this means is that the next `RETURN` statement executed will exit to a `GOSUB` one higher in the hierarchy than otherwise would have been the case. The normal use for this would be in a subroutine in which you wish to transfer control to a main level error routine.

Example:

```
:SUB1
  ON TIMEOUT 10 GOTO SUB1ERR
  RECEIVE "Name: "
  SNDLINE $LOGIN
  RETURN
:SUB1ERR
  POPUP
  GOTO ERR
```

If the `POPUP` had not been executed here, then you would still effectively be 'in' the subroutine, and the next `RETURN` executed would take you to the place where `SUB1` was called. `POPUP` removes the reference to `SUB1` from the stack of `GOSUB` references.

The PORT command

Syntax: `PORT <number>`

This command defines which communications port to use (if you don't use the command, then of course Glink will just continue on whichever port was in use at the time the script was started).

The POS command

Syntax: `POS <row> <column>`

This command simply moves the cursor to the specified position, and is useful in cases where you wish to present a particular message in a predefined place on the screen.

The PPROFILE command

Syntax: `PPROFILE <filename> <section> <item> <value>`

This command allows you to write options to INI files that are formatted in the same way as for example the `SYSTEM.INI` file, with sections introduced using headings in the format

```
[section-name]
```

The command writes the value of the selected option into the file. If necessary the file will be created and the chosen section inserted. For example, if you wanted the `[boot.description]` section of your `SYSTEM.INI` file to contain the following line:

```
codepage=850
```

then executing the following script command:

```
PPROFILE "C:\WINDOWS\SYSTEM.INI"
   "boot.description" "codepage" "850"
```

would do this for you.

The PREMOTE command

Syntax: `PREMOTE <parameter number> <parameter value>`

This command has relevance only on the Atlantis and Eicon NABIOS interfaces for X.25 connections. It allows you to control remote PAD parameters from a script that is handling X.25 logins (for setting of local PAD parameters, see the `PSET` command below). The command sets the equivalent PAD parameter for the terminal logging into Glink - for example, you can turn off echo at the other end by using:

```
PREMOTE 2 0
```

The PRINT command

Syntax: `PRINT <string>`

The `PRINT` command will use the built-in printing routines to send the specified data to the printer. For example, to send a form feed character to the printer you would write

```
PRINT "^L"
```

The PSET command

Syntax: `PSET <parameter number> <parameter value>`

This command has relevance only on the Atlantis and Eicon NABIOS interfaces for X.25 connections, and allows you to control local PAD parameters from a script (check documentation from your PTT for more information about PAD parameters).

The PUTFILE command

Syntax: `PUTFILE <protocol> <filename>`

The opposite of the `GETFILE` command, this will start a file transfer from your machine to the host machine. For example, to send a file to the host using KERMIT you would use:

```
PUTFILE KERMIT "FILENAME"
```

For the Kermit protocol only, the file name may be specified in the format `"PC-name;Host-name"`, in which case the name used for the host machine may be different from the name used locally on the PC. Note that in the case of FTP transfers you should use the `FTP` script command.

In the case of an FTP transfer, the file name **must** be coded with the local and remote names. Additionally you must have set the host configuration name and/or parameters using SET FTPSERVER so as to provide the necessary connection information.

See the `GETFILE` command for a list of valid file transfer protocol keywords.

The QUIT command

Syntax: `QUIT`

This stops script processing and returns the emulator to interactive mode.

The RATR command

Syntax: `RATR <%var> <row> <column>`

This command allows you to read the logical and physical attributes from a given screen position. The result returned in the `%var` variable consists of a 48-character string of ones and zeros. Each of these indicates the presence or absence of a particular attribute. The first 16 relate to the actual appearance of the character position on the screen, after any user-specified mappings have been processed. The last 32 relate to attributes set by the host machine - these include not only visual attributes but also logical attributes. Note that for simple character-based emulations (ANSI, for example) only the first 16 positions are valid.

For example, the following script fragment checks to see whether the character at the current cursor position is underlined (on the screen) or not:

```
RATR %1 $Y $X
SUBS %1 %1 9 1
IF (%1 EQ "1") GOTO Underlined
```

A complete list of the various positions and their meanings follows:

Pos	Type	Meaning		Pos	Type	Meaning
1	Physical	Reserved		25	Logical	Reserved
2	Physical	Double-width line		26	Logical	Reserved
3	Physical	Reserved		27	Logical	Digit
4	Physical	Sixel graphics		28	Logical	Numeric
5	Physical	Double height lower		29	Logical	Alphabetic
6	Physical	Blink		30	Logical	Right justify
7	Physical	Double width		31	Logical	Must fill
8	Physical	Double height upper		32	Logical	Must enter
9	Physical	Underline		33	Logical	Attribute start
10	Physical	Background red		34	Logical	Unprotected
11	Physical	Background green		35	Logical	Transmittable
12	Physical	Background blue		36	Logical	Modified
13	Physical	Bold		37	Logical	Omit print
14	Physical	Foreground red		38	Logical	Double width
15	Physical	Foreground green		39	Logical	Double height
16	Physical	Foreground blue		40	Logical	Reserved
17	Logical	Reserved		41	Logical	Underline

Pos	Type	Meaning		Pos	Type	Meaning
18	Logical	Reserved		42	Logical	Inverse
19	Logical	Reserved		43	Logical	Hidden
20	Logical	Reserved		44	Logical	Blink
21	Logical	Reserved		45	Logical	Low intensity
22	Logical	Reserved		46	Logical	Foreground red
23	Logical	Reserved		47	Logical	Foreground green
24	Logical	Reserved		48	Logical	Foreground blue

The RCVLINE command

Syntax: `RCVLINE <%var> <max length>`

This command collects a line of data from the host into the specified script variable. The maximum length of data to receive must also be specified. For example:

```
RCVLINE %4 12
```

would collect a maximum of 12 characters into script variable number 4. The receive is terminated either when the required number of characters has been received or when a terminator is received (default terminators are ETX, EOT and CR). Any line feed characters received are stripped, as are NULs, DELs and ETBs in order to make it easier to receive multiple variable length strings from the host.

For more advanced applications, you may wish to modify the way that RCVLINE filters characters and also wish to consider characters other than the defailts to be terminators. This is provided for by the SET RLFILTER and SET RLTERM commands. For example, you may wish to set things up so that the ETB character acts as a terminator rather than be filtered out. You can do this by specifying

```
SET RLFILTER "^$0A^$00^$7F"
```

and

```
SET RLTERM "^$0D^$03^$04^$17"
```

before your RCVLINE command. Note that you may check which character actually caused the receive operation to terminate by inspecting the contents of the built-in $RLTERM variable (in the case where the receive terminates because the specified number of characters were received this variable will be empty).

The RCVTURN command

Syntax: `RCVTURN`

This command waits for the emulator to receive the ‘turn’ from the host. This only has a definitive meaning for those interfaces that support it (G&R Ggate and DGA, TNVIP, TN3270 and the X.25/TGX interfaces). On other interfaces the command will simply wait for a pause in the data sent from the host. This may be used to check that the host has finished sending data, for example in order to inspect the contents of the form that has been received.

The RD command

Syntax: `RD <directory name>`

The `RD` command allows you to delete a directory (subject to the normal rules about deleting directories) without having to use the `DOS` command. `IF OK` may be used to test whether or not the directory was deleted successfully. For example:

```
RD "\TEMP"
```

The RDIAL command

Syntax: `RDIAL [<max attempts>]`

This command will start a 'queued' redial of all marked numbers in the dial directory, in just the same way as pressing the `NUM/+` key interactively while in the dial directory does. As soon as a connection has been made with any of the marked hosts, execution continues on the statement after the `RDIAL` command. If no hosts have been marked then this will be reflected in the script `OK` variable. You may use this fact to program a 'loop' that processes marked entries, using the fact that the mark is removed for each site as it is contacted.

An optional parameter provides for termination of the redial after a given number of unsuccessful attempts:

```
RDIAL 10
```

will either return a connection to one of the marked hosts, return immediately if no hosts are marked, or return after 10 consecutive attempts to dial have failed. You may distinguish between these three different cases using the `IF FAIL` command. `OK` is set true only when a connection is established, while `FAIL` is set true if the return from `RDIAL` is because of failure to contact a host rather than an absence of marked host machines. The following should make this clear:

```
RDIAL 10
IF OK   GOTO CONNECTED
IF FAIL GOTO NOCONTACT
GOTO NONEMARKED
```

The RECEIVE command

Syntax: `RECEIVE <string>`

This command specifies a string for which the script is to wait. The script will only wait for as long as is defined in the `TIMEOUT` command (default is 60 seconds) and then either go to the `ERRORGOTO` label if one has been defined, or the next command if not. While the script is waiting for the string to arrive, patterns will also be checked and may be tested for afterwards with the `IF` command. However, unlike `MATCH`, `RECEIVE` will not continue just because a pattern was found - it waits until the actual string you tell it to wait for has arrived. For example:

```
RECEIVE "Password:"
```

The `RECEIVE` command is by default case sensitive, but this may be changed using the `SET CASE OFF` command. If a `SET IDLE` command is active then Glink will wait for the line to be idle for the specified time before continuing to execute the script (this may be found necessary on half-duplex lines). This is obviously not a completely reliable way of knowing that the complete output has been received.

If you are using a communications protocol that supports the concept of 'turn' i.e. a signal that says when the mainframe has finished sending to the terminal, then you can wait for your 'turn' by issuing:

```
RECEIVE "^$03"
```

ETX (0x03) is the internal Glink signal for turn, and is delivered by Ggate, DGA, TNVIP and TN3270, although IBM applications sometimes deliver the 'turn', then change their mind and send more (also the 0x03 may be delivered as part of normal transmitted data). Note that you may use the `RCVTURN` command when you are not waiting for any particular string, but would like to know when the mainframe is finished so that the script can process the data received.

The RECS command

Syntax: `RECS <string>`

This command is similar to the `RECEIVE` command, with the exception that the characters specified need not be received together; any characters may be received between the characters specified in the string to be received. This is especially useful when 'invisible' characters are being sent on the communications line between the characters you can actually see on the screen. For example, if the host is actually sending the sequence:

```
Enter your id please<cr><lf><del><del>?
```

then you could tackle this (without starting to specify control characters directly) either by using two `RECEIVE` statements like this:

```
RECEIVE "Enter your id please"
RECEIVE "?"
```

or by specifying just one `RECS` statement like this:

```
RECS "Enter your id please?"
```

The `RECS` command is by default case sensitive, but this may be changed using the `SET CASE OFF` command. If a `SET IDLE` command is active, then Glink will wait for the line to be idle for the specified time before continuing script execution. You may find you need this on half-duplex lines.

The REMENU command

Syntax: `REMENU`

If the action performed by some option on a menu does not actually result in removal of the menu from the screen, then you may find yourself in the situation that you wish to retain the current menu on the screen and let the user choose another option from the same menu. Typically, this would be execution of a DOS command or some kind of file processing. This is done with the `REMENU` command. Note that it is your own responsibility to ensure that the command is executed while the menu in question is still on the screen, and that you do NOT need to execute any of the `MENU`, `MOPC`, `MOP`, `MTEXT` or `DOMENU` commands associated with the original menu. The system remembers all the parameters associated with the last menu to be executed and simply sets them up again without presenting the menu on the screen again.

The REN command

Syntax: `REN <filename> <new name>`

The `REN` command allows you to rename a DOS file without having to use the DOS command. `IF OK` may be used to test whether or not the command was successful. For example:

```
REN "MYFILE" "HISFILE"
```

The REPLACE command

Syntax: `REPLACE <%var> <from> <to>`

This command scans the specified variable for occurrences of the `<from>` string and replaces each instance with the `<to>` string. If this would result in a loop (because the `<from>` string is contained in the `<to>` string) then no action is taken. For example:

```
ASSIGN %1 "I can walk the walk"
REPLACE %1 "walk" "talk"
```

would produce "`I can talk the talk`" in variable `%1`. The replacement is repeated as many times as necessary, for example:

```
ASSIGN %1 "1,,,2,,3,,,,,4"
REPLACE %1 ",," ","
```

would give "`1,2,3,4`" in variable `%1`.

The RESET command

Syntax: `RESET <event type>`

This command resets the effect of an active `ON` statement. Available values for `<event type>` are:

```
INACTIVITY
KEY
OFFLINE
SESSION
TIMEOUT
TURN
```

Note that `RESET KEY` resets `ON` actions for all keys. To reset a single action use the `RKEY` command. `RESET TIMEOUT` resets not only the `ON TIMEOUT` action, but also sets the `TIMEOUT` value back to its default of 60 seconds.

The RETCALL command

Syntax: `RETCALL`

This command allows you to return directly to a calling script, even when you are in a routine several 'levels' down because of `GOSUB`s in the current script. Normally you use `RETURN` to exit from both GOSUB and CALL, but this command allows you to 'ignore' `GOSUB`s that are still active since the last `CALL`.

The RETURN command

Syntax: `RETURN`

This command returns control either from a procedure that was called with a `GOSUB` command or from a script file that was called with the `CALL` command. (If no `GOSUB` or `CALL` is active, then `RETURN` will terminate script execution). In the case where a script executes all the way to the end of a script file, an automatic `RETURN` will be done for you after the last statement in the script file has been executed.

The RFORM command

Syntax: `RFORM <%var> <number>`

This command allows you to read a field from a VIP form that is on-screen when the command is executed. For example:

```
RFORM %3 12
```

This reads the 12th variable field from the currently active form and returns the result in the `%3` variable. If `<number>` is larger than the number of fields in the form then a null string will be returned. If `<number>` is zero then the current field will be returned, defined as the field containing the cursor (if the cursor is positioned in a variable field) or the field immediately preceding the cursor (if the cursor is not currently inside a variable field).

The RKEY command

Syntax: `RKEY <keyname>`

The `RKEY` command resets an action that has been specified with an `ON KEY` keyname statement (to reset all such actions use `RESET KEY`). `<keyname>` is specified using the same mnemonics as are used for the `ON KEY` and `KEYS` statements (a list is provided in the description of the `KEYS` statement).

The ROLL command

Syntax: `ROLL {ON|OFF}`

This command allows you to turn 'roll mode' on and off. A terminal operating in 'roll mode' will scroll the picture off the screen when receiving data past the bottom of the screen. A terminal operating with roll mode disabled will give an error if a program attempts to write past the end of the screen.

The RSBK command

Syntax: `RSBK <%var> <row> <column> <length>`

This command allows you to 'read' a string of characters directly from the scrollback buffer. `<row>` in the above refers to the row number in the scrollback buffer, where row **zero** is the first row that has scrolled off the screen, row **one** is the row that scrolled off prior to that, and so on. In other words, the higher the row number, the earlier the row. If you try to access a row that has scrolled out of the scrollback buffer then the `RSBK` command will return a null string. Accessing the scrollback buffer with a negative line number will access the corresponding line in the current screen.

A 'side-effect' of the `RSBK` command when used in the Windows versions is that any mark that may be in the scrollback window will be removed.

The following Windows example uses the internal variables providing information about a marked area in the scrollback buffer in order to extract the actual marked data and write it to a file. Note that the 'top' coordinate returned by `$MSY1` will in practice be larger than the 'bottom' coordinate, in that the scrollback buffer is numbered in reverse order.

```
IF ($MSX1 EQN 0) GOTO NoMark
ASSIGN %1 $MSX2
SUBTRACT %1 $MSX1
ADD %1 1
ASSIGN %2 $MSY1
ASSIGN %3 $MSY2
ASSIGN %4 $MSX1
FOPEN #1 APPEND "SAVE.TXT"
WHILE (%2 GEN %3)
  RSBK %5 %2 %4 %1
  FWTLINE #1 %5
```

```
    SUBTRACT %2 1
  ENDWHILE
  FCLOSE #1
  RETURN
:NoMark BEEP
```

The RSCR command

Syntax: `RSCR <%var> <row> <column> <length>`

This command allows you to 'read' a string of characters directly from the screen at a given position. The characters on the screen at the given location are placed into the defined script variable. If you wish to use this command to read data from the status line you may do so, but must specify a row number of zero rather than the actual row as counted on the screen.

A 'side-effect' of the command when used in the Windows versions is that any mark that might currently be on the screen will be removed.

The following Windows example uses the internal variables providing information about marked areas on the screen in order to collect the data from the screen and write it to a file:

```
  IF ($MX1 EQN 0) GOTO NoMark
  ASSIGN %1 $MX2
  SUBTRACT %1 $MX1
  ADD %1 1
  ASSIGN %2 $MY1
  ASSIGN %3 $MY2
  ASSIGN %4 $MX1
  FOPEN #1 APPEND "SAVE.TXT"
  WHILE (%2 LEN %3)
    RSCR %5 %2 %4 %1
    FWTLINE #1 %5
    ADD %2 1
  ENDWHILE
  FCLOSE #1
  RETURN
:NoMark  BEEP
```

The SCAN command

Syntax: `SCAN <string> <string>`

This is used to check whether a given string contains another one. The `OK` variable (see the `IF` command) is used to check whether it did or not. For example:

```
SCAN %5 "A"
IF OK GOTO FOUND
```

checks whether or not script variable number 5 contains an uppercase A. Additionally, the `$SCAN` internal variable will contain the offset at which the string was found (or zero when it was not found).

The SCREEN command

Syntax: `SCREEN {ON|OFF}`

The `SCREEN OFF` command may be used to turn off ALL screen updating, including file transfer windows, status line updates, and the like. Updating is re-enabled with `SCREEN ON`. The screen will in any case be turned on when the script completes; also if any local key is pressed. This is to ensure that inadvertent use of a `SCREEN OFF` command without an accompanying `SCREEN ON` doesn't confuse you completely. The screen will still be updated in background and `SCREEN ON` will restore the current background image when used. If you need to turn the screen off more permanently, with absolute control over when it is turned on again, use the `SET UPDATE` command instead.

The SECURE command

Syntax:
`SECURE{SCROLLBACK|LINEBUFFER|EDIT|ALL}{ON|OFF|ERASE}`

The `SECURE` command is provided in order that a script may prevent access to data that otherwise might represent a potential security risk. The data referred to in the above syntax relates to the scrollback buffer used for saving data that has scrolled off the screen, the line buffer that shows raw data and that may be accessed from the line menu, and the edit buffer that may be used for command recall. For each of the above, the `ERASE` option will remove the entire contents of the buffer, while `ON` and `OFF` toggle logging of data into the buffer. The `ALL` option will perform the desired action for all three buffers simultaneously. For example, to erase everything that the system has logged so far you could use:

```
CLEAR
SECURE ALL ERASE
```

The `CLEAR` in the above is used in order to remove data from the current screen, which would otherwise possibly scroll off into the scrollback buffer at some later stage. If the `SECURE LINEBUFFER ON` command is used this will also disable use of debug mode.

The SEND command

Syntax: `SEND <string>`

This is used to send a defined string of characters to the host, with no terminator. For example:

```
SEND "L FRED^M"
```

If you are working in VIP TEXT or FORMS mode then the string will not be sent as such. Instead it will be given to the emulation procedures that will perform the transmission when `SNDLINE` is used or the transmit key is pressed. To actually send a string directly to the communications line in such a situation, use the `TRANSMIT` command.

The SEPMENU command

Syntax: `SEPMENU`

This command adds a horizontal separating line between menu items built with the `BUILDMENU` and `ADDMENU` script commands. It should be placed between the `ADDMENU` statements for those items you wish to separate.

The SERVER command

Syntax: `SERVER {FINISH|LOGOUT|START|<character> <parameter>}`

This is normally used to terminate a dialog with a Kermit server. One of the first two commands would typically be used after a number of `GETFILE KSERVE` and `PUTFILE KSERVE` commands. You may also issue 'generic' server commands using the form:

```
SERVER <character> <parameter>
```

Here `<character>` is the identifier for the generic command. For example, if the remote server supports the generic 'erase' command you could erase a file using:

```
SERVER "E" "MYFILE.EXT"
```

You may also start a local Kermit server using the command:

```
SERVER START
```

The SET command

Syntax: See below

The `SET` command is a general-purpose command with many options that may be used to set various configuration parameters. More detailed explanations of each of these will be found in the *Configuring the emulator* chapter in the *Administrator's Guide*. Here we will just give an indication of which parameter is being set. The following `SET` commands are currently implemented:

`SET 3D {ON|OFF}`
Toggles the setting for the 3D variable effect.

`SET 3D COLORS {GLINK|WINDOWS}`
Toggles the setting for use of Glink or Windows colors when using 3D variables.

`SET 3D CARET {GLINK|WINDOWS}`
Toggles the setting for use of Glink or Windows caret when using 3D variables.

`SET 3D FIXDKU {ON|OFF}`
Toggles the 'fix DKU attributes' option.

`SET 3D UNDERLINE {ON|OFF}`
Toggles the setting for stripping of underline attributes when using 3D variables.

`SET ALTERNATE <servername>`
Sets the name for the alternate server on Ggate connections.

`SET ASCII CRCAPTURE {ON|OFF}`
Toggles the 'capture delimiter' option in the file transfer setup menu.

`SET ASCII EXPAND {ON|OFF}`
Toggles the 'ASC expand blanks' option in the file transfer setup menu.

`SET ASCII HANDSHAKE <character>`
Sets the 'ASC handshake char' in the file transfer setup menu (the character must be specified as a decimal number).

`SET ASCII LPACE <tenths>`
Sets the 'ASC line pacing' value in the file transfer setup menu.

`SET ASCII PACE <milliseconds>`
Sets the 'ASC pacing' value in the file transfer setup menu.

`SET ASCII XLCR {STRIP|CR|LF|CRLF}`
Sets the 'ASC CR translation' option in the file transfer setup menu.

`SET ASCII XLLF {STRIP|CR|LF|CRLF}`
Sets the 'ASC LF translation' option in the file transfer setup menu.

SET BREAK {ENABLE|DISABLE}
Enables or disables the ability to send a break to the host.

SET BUTTON {ON|OFF}
Turns the button bar on or off.

SET BUTTON FONT "fontname" fontsize
For the Windows versions only, sets the font used for buttons on the button bar created with the script BUTTON command.

SET BUTTON HELP n "help text"
For the Windows versions only, sets the help text for a button on the button bar created with the script BUTTON command.

SET BUTTON {NUMBER|ROWS} <number>
For the Windows versions only, sets the number of buttons (or the number of rows) in the button bar, containing buttons set with the script BUTTON command. See the description of the BUTTON command for more information, on page 38.

SET CAPS {ON|OFF}
Sets or resets the CAPS LOCK keyboard status. Note that this is not supported on Win95/98/ME platforms.

SET CAPTION {ON|OFF}
Turns the caption bar on or off.

SET CASE {ON|OFF}
Determines whether or not comparisons should be case sensitive (default is ON). This affects all commands that compare strings, including patterns and received strings. Note that the current case sensitivity setting may be retrieved from the $CASE built-in variable.

SET COLOR SCREEN <color>
SET COLOR STATUS <color>
Sets the basic emulator screen colors and the colors for the status line

Valid syntax for <color> in the above is:
[BRIGHT] foreground ON background
where foreground and background are chosen from:
BLACK BLUE GREEN CYAN RED MAGENTA YELLOW WHITE
For example:
SET COLOR SCREEN BRIGHT WHITE ON BLUE.

`SET COMWARNING { ON | OFF }`
Controls whether or not serial port conflicts with other applications should give a warning message (this will also depend on your Windows settings otherwise). Typically, this command would be used to disable the message from Windows saying that the port is in use by another application.

`SET CONNECT {ENABLE|DISABLE}`
Enables or disables the ability to connect to a host.

`SET CTABS {ON|OFF}`
Controls the 'clipboard with tabs' option in the menu bar.

`SET DBGFILE filename`
Specifies the name for the line debug file (see DEBUG ON/OFF). This command only has an effect the next time line debugging is turned on (if line debugging is already on it will continue on the current line debugging file).

`SET DEBUGFILE filename`
Specifies the name for the debug file (see STRACE ON/OFF). This command only has an effect the next time debugging is turned on (if debugging is already on it will continue on the current debugging file).

`SET DELIMITERS delimiters`
Sets delimiters for separation of words on-screen (this may also be set in the screen options configuration). These delimiters are used to decide how marking words in the main screen and scrollback screen will be done when they are double-clicked, and will also affect the behaviour of the `GETWORD` script command. Current delimiters are available in the `$DELIMITERS` variable.

`SET DEVICE "devicename"`
Sets the LU device name for those TN3270 or TN5250 servers that support selection of device by name.

`SET DISCONNECT {ENABLE|DISABLE}`
Enables or disables the ability to disconnect from a host.

`SET DKUCOLOR {1M|4A|4B|7Q|7G}`
Sets the color mode for the DKU emulation.

`SET DKUMODEL {7107|7211}`
Sets the terminal model for the DKU emulation.

`SET DOSSHOW {<option>}`
Specifies how a program started with the DOS command should be displayed. The same options are available for this command as are provided for the `WINDOW` command.

`SET DOSWAIT {YES|NO}`
Specifies whether Glink should wait for an external command to complete before continuing. When an external program is started (using the DOS script command) from a script run by the Windows versions, the script will continue running in parallel with the external program. Setting this option will stop Glink from running until the other window has been closed.

`SET DYNAMIC rows columns`
Sets the IBM-DYNAMIC terminal model and specifies rows and columns for the alternate screensize.

`SET EXIT {ENABLE|DISABLE}`
Enables or disables the ability to terminate the emulator.

`SET FCLICK X1 Y1 X2 Y2 "url"`
Defines an area of the frame wallpaper that may be clicked. The X and Y coordinates relate to the original (unstretched) bitmap coordinates.

`SET FKEYBAR {ON|OFF}`
Turns the function key bar on or off.

`SET FOCUS {ENABLE|DISABLE}`
Enables or disables the ability to switch to the Glink window.

`SET FONT <height> <width>`
Sets the font size (in pixels) for the current Glink window.

`SET FRAME {LEFT|RIGHT|TOP|BOTTOM} <size>`
Sets the size of the frame around the emulation window

`SET FRAME {PIXELS|PERCENT}`
Specifies whether the frame size is in pixels or as a percentage of the main window.

`SET FRAMEPAPER filename`
Sets the name of the file to use as wallpaper around the emulation frame.

`SET FTP ADDRESS host_address`
Sets the host IP address to be used for ftp transfers using the GlinkFTP client. The built-in variable `$FTPADDRESS` returns the current setting for this option.

`SET FTP ASCII`
Sets ASCII mode for ftp transfers using the GlinkFTP client. The built-in variable `$FTPTRANSFER` returns a value of 1 when this option is set.

`SET FTP AUTO`
Sets automatic selection of transfer mode (based on file extension) for ftp transfers using the GlinkFTP client. The built-in variable `$FTPTRANSFER` returns a value of 0 when this option is set.

`SET FTP BINARY`
Sets BINARY mode for ftp transfers using the GlinkFTP client. The built-in variable `$FTPTRANSFER` returns a value of 2 when this option is set.

`SET FTP CONFIG filename`
Sets the configuration file name to be used for ftp transfers using the GlinkFTP client. The built-in variable `$FTPCONFIG` returns the current setting for this option.

`SET FTP DEFAULT hostname`
Sets the host name (as configured in GlinkFTP) to be used as the default host for ftp transfers using the GlinkFTP client. The built-in variable `$FTPDEFAULT` returns the current setting for this option.

`SET FTP HOST hostname`
Sets the host (as configured in GlinkFTP) to be used for ftp transfers using the GlinkFTP client. The built-in variable `$FTPHOST` returns the current setting for this option.

`SET FTP LOCAL8`
Sets LOCAL8 mode for ftp transfers using the GlinkFTP client. The built-in variable `$FTPTRANSFER` returns a value of 3 when this option is set.

`SET FTP NORM`
Resets passive mode for ftp connections using the GlinkFTP client. The built-in variable `$FTPMODE` returns a value of 0 when this option is set.

`SET FTP NOWAIT`
Tells the script that it may continue execution after starting an FTP transfer. The transfer continues in parallel with script execution, and you may use the `$FTP` built-in script variable to check on its current status (a return of zero means that the transfer is complete).

`SET FTP PASSWORD password`
Sets the login password for ftp transfers using the GlinkFTP client. The built-in variable `$FTPPASSWORD` returns the current setting for this option.

`SET FTP PASV`
Sets passive mode for ftp connections using the GlinkFTP client. The built-in variable `$FTPMODE` returns a value of 1 when this option is set.

`SET FTP SILENT`
Sets silent mode (no messages or display) for ftp transfers using the GlinkFTP client. The built-in variable `$FTPSILENT` returns a value of 1 when this option is set.

`SET FTP USER username`
Sets the login user name for ftp transfers using the GlinkFTP client. The built-in variable `$FTPUSER` returns the current setting for this option.

`SET FTP VERBOSE`
Turns off silent mode for ftp transfers using the GlinkFTP client. The built-in variable `$FTPSILENT` returns a value of 0 when this option is set.

`SET FTP WAIT`
Tells the script to wait for FTP transfers to complete before continuing (this is the default).

`SET IBMMODEL model_name`
Sets the IBM model name.
Valid IBM3270 models are `3279-2, 3279-3, 3278-1, 3278-2, 3278-3, 3278-4, 3278-5, 3279-2E, 3279-3E, 3278-1E, 3278-2E, 3278-3E, 3278-4E, 3278-5E, DYNAMIC` and (printer) `3278-1`.
Valid IBM5250 models are IBM5250 model `3179_2, 3180_2, 3196_A1, 3477_FC, 3477_FG, 5251_11, 5291_1, 5292_2, 5555_C01, 5555_B01, 3812_1` and `5553_B01`.
Valid IBM3151 models are `3151_11, 3151_31, 3151_41, 3151_51` and `3151_61`.

`SET IDLE <seconds>`
Specifies the time (in seconds, tenths of seconds may be specified). Glink should wait for the line to be idle after a match is found when using the `CONVERSE`, `MATCH`, `RECEIVE` or `RECS` commands. Depending on the setting of `SET IPATTERN` at the time the pattern was set, the delay may also be applied before execution of `WHEN` statements when patterns have been matched. This may be found useful when programming scripts that talk to hosts that require an additional delay before new input is sent, or when the host appends additional characters to the end of the string being waited for. Note that the built-in `$IDLE` variable provides you with the current value of the idle timer.

`SET IPATTERN {ON | OFF}`
Specifies whether the `SET IDLE` timer should be applied before executing the `WHEN` statement when a pattern is matched. The default is `OFF` (in other words the timer will be applied), but for compatibility with some earlier releases, which did not wait in this situation, you may find that `SET IPATTERN ON` is needed. The setting is saved on a per-pattern basis and the setting that applies to any particular pattern is that which applied when the pattern itself was set.

`SET INPUT {ENABLE|DISABLE}`
Enables or disables input into the Glink window. Note that if you disable input in this way then your script must reenable it before terminating or the Glink window will be locked.

`SET INSID "name"`
Sets the host installation ID for DNTD interfaces.

`SET ISTATUS {AUTO | SHOW | HIDE}`
Specifies how the status of a file operation that accesses the internet should be shown. The default (which will be set after each such operation) is to not show a status window until a transfer has been running for at least 5 seconds. If you set it to SHOW then the status window will appear immediately, whereas if you set it to HIDE then the status window will never be shown. Note that if the window is shown then the user will have the opportunity to interrupt the transfer, and if you wish to avoid this you should use HIDE. Note that the current status key is available in the $ISTATUS built-in variable

`SET KBDBAR {ON|OFF}`
Turns the keyboard bar on or off.

`SET KERMIT ADDCTRLZ {ON|OFF}`
Toggles the 'Add Control-Z' option in the Kermit setup menu.

`SET KERMIT CMP8 {ON|OFF}`
Toggles the 'Enable DPS8 compression' option in the Kermit setup menu.

`SET KERMIT EOFCTRLZ {ON|OFF}`
Toggles the 'Ctrl-Z means end of file' option in the Kermit setup menu.

`SET KERMIT FORCEFTRAN {ON|OFF}`
Toggles the 'Non-standard FTRAN' option in the Kermit setup menu.

`SET KERMIT NOXLATE {ON|OFF}`
Toggles the 'No file name translation' option in the Kermit setup menu.

`SET KERMIT OVERRIDE {ON|OFF}`
Toggles the 'Override host packet size' option in the Kermit setup menu.

`SET KERMIT PACE <milliseconds>`
Sets the 'Kermit pacing' option in the Kermit setup menu.

`SET KERMIT PACKETSIZE <number>`
Sets the 'Maximum long packet size' parameter in the Kermit setup menu.

`SET KERMIT QUOTE <character>`
Sets the 'Kermit host quote' option in the Kermit setup menu.

`SET KERMIT RETRIES <number>`
Sets the 'Kermit retries' option in the Kermit setup menu.

`SET KERMIT SOH <character>`
Sets the 'Kermit packet header' option in the Kermit setup menu. The character must be specified as a decimal number.

`SET KERMIT TABXPAND <number>`
Sets the 'Tab expansion' option in the Kermit setup menu.

`SET KERMIT TIMEOUT <number>`
Sets the 'Kermit timeout' option in the Kermit setup menu.

`SET KERMIT WINDOWSIZE <number>`
Sets the 'Maximum window size' in the Kermit setup menu.

`SET KEYBOARD {ON|OFF}`
Allows you to disable the keyboard completely. Keystrokes typed while `SET KEYBOARD OFF` is in effect will not be read by the emulator but left in the keyboard input buffer until `SET KEYBOARD ON` is executed (or until you leave the script or exit from Glink)

`SET LOOPCHECK {ON|OFF}`
There is an internal loop detection routine inside the script module, which attempts to ensure that a tight loop inside a script does not totally disable the emulator. In certain cases, this can cause undesirable side effects, typically in cases where there is a large amount of menu interaction with the user with no line activity. This command allows you to disable the checking.

`SET MENU {ON|OFF}`
Turns the menu bar on or off.

`SET MARKMODE {OLD|NEW}`
Controls the function of the `MARK` command.

`SET MODE APCURSORPAD {ON|OFF}`
Toggles the 'Cursor application mode' option in the toggles menu.

`SET MODE APKEYPAD {ON|OFF}`
Toggles the 'VT100 keypad mode' option in the toggles menu.

`SET MODE AUTOLF {ON|OFF}`
Toggles the 'Auto LF out' option in the toggles menu.

`SET MODE DESTBS {ON|OFF}`
Toggles the 'Destructive backspace' option in the Emulator setup menu.

`SET MODE GRAPHICS {ON|OFF}`
Toggles the 'Graphics mode' option in the toggles menu.

`SET MODE INSERT {ON|OFF}`
Toggles the 'Insert mode' option in the toggles menu.

`SET MODE SSM {ON|OFF}`
Toggles the 'Space suppression' option in the toggles menu.

`SET MODE TYPEAHEAD {ON|OFF}`
Toggles the 'Typeahead mode' option in the toggles menu.

`SET MODE VIP {CHAR|TEXT|FORM|TXRT}`
Sets the VIP emulation mode to CHARacter, TEXT, FORMs or TX-RET mode.

`SET MODE WRAP {ON|OFF}`
Toggles the 'Autowrapping' option in the toggles menu.

`SET MRECT {ON|OFF}`
Toggles the 'Mark Rectangles' option in the edit menu.

`SET NOISELEVEL <number>`
Sets the 'Noise level' option in the General setup menu.

`SET NOREMOTE {ON|OFF}`
Toggles the 'Ignore remote commands' option in the General setup menu.

`SET NUML {ON|OFF}`
Sets or resets the NumLock keyboard status.

`SET PRINTER <printer_name[,driver,port]>`
Changes the current printer and sets the `OK` variable. Note that the current printer is available in the `$PRINTER` built-in variable.

`SET PROCESS {ON|OFF}`
Determines whether input from the line should be processed by the emulator while the script is running.

`SET RELINQUISH {ON|OFF}`
Determines whether the script processor should release control to other applications while executing. The default for this option is ON, but you may wish to turn it OFF when executing time-critical code that has to run as fast as possible. The effect of turning the relinquish option OFF will vary from machine to machine.

`SET RESIZE {NONE|FONT|WINDOW}`
Specifies how the window should react when resized. NONE changes the displayed area (using scrollbars), FONT will change the font size to fit in the current window, while WINDOW will change the number of rows and columns displayed.

`SET RESOURCE <resource_name>`

Sets the TNVIP resource or DSA/DIWS host profile name for the communications interface. Note that the current server name is available in the $RESOURCE built-in variable. If you wish to use the defined DSA/DIWS profile for a particular host but override one or more of the host parameters then you may do this by including the relevant parameters after the host name. For example:

```
SET RESOURCE "DPS8 -DNPH02"
```

For DNTD it will set the host INSID, while for X.25 and similar services it may be used to specify the name of the communications server used to reach the host.

`SET RLFILTER characters`
Controls filtering for the script RCVLINE command. Characters specified with this option will be stripped from the data returned by the command. The default setting for the option is LF, NUL, DEL and ETB.

`SET RLTERM characters`
Controls termination characters for the script RCVLINE command. Characters specified with this option will cause the receive operation to terminate. The default setting for the option is CR, ETX and EOT.

`SET ROUND {ON|OFF}`
Controls whether fractional numbers are rounded or truncated when used in integer contexts and when using the script `TRUNCATE` command. Note that the current rounding status is available in the `$ROUND` built-in variable.

`SET SCREEN LENGTH <number>`
Sets the number of lines for the current emulation. Only up to the maximum number of lines specified in the `/Rnn` startup parameter may be used.

`SET SCREEN ONTIME {ON|OFF}`
Toggles the 'Clock shows time online' option in the Modem setup menu.

`SET SCREEN SBSAVE {ON|OFF}`
Toggles the 'CLR save in scrollback' option in the Screen setup menu.

`SET SCREEN UPDATE {DIRECT|RETRACE|BIOS}`
Sets the 'Screen Update' option in the Screen setup menu.

`SET SCREEN WIDTH <number>`
Sets the number of columns to use on the emulation screen. Any number of columns between 40 and 132 may be specified.

`SET SERVER <server_name>`
Sets the server name or IP address for the TCP communications interface. Note that the current server name is available in the `$SERVER` built-in variable. Note also that on X.25 and similar interfaces this command will set the X.25 address or equivalent; if you actually intend to change the name of the comms server then the `SET RESOURCE` command should be used.

`SET SHARE {COMPATIBILITY|READ|WRITE|RW|EXCLUSIVE}`
Specifies the sharing mode for the next FOPEN script command.

`SET SILENT {ON|OFF}`
Toggles the 'Silent mode' option in the General setup menu.

`SET SPATH {ON|OFF}`
Sets DOS command execution to perform searching in the PATH before execution via `COMMAND.COM` (see the DOS command).

`SET SSH PASSWORD <password>`
Sets the password for the SSH server.

`SET SSH PKEY <keyname>`
Sets the private key used in the SSH PuTTY interface. Note that the current private key is available in the `$PKEY` built-in variable.

`SET SSH SERVER <name>`
Sets the server name for the SSH server.

`SET SSH USER <name>`
Sets the user name for the SSH server

`SET SSL [ENABLE|DISABLE]`
Enables or disables use of secure sockets.

`SET SSL CLIENT AUTHENTICATE [ENABLE|DISABLE]`
Enables or disables client authentication for secure sockets.

`SET SSL CLIENT CERTIFICATE <name>`
Sets the name of the client certificate to use for client authentication.

`SET SSL KEYEXCHANGE [AUTO|RSA|DH]`
Sets the key exchange protocol to use for secure socket connections.

`SET SSL PROTOCOL [AUTO|PCT1|SSL2|SSL3|TLS1]`
Sets the protocol to use for secure socket connections.

`SET SSL SERVER VALIDATE [ENABLE|DISABLE]`
Enables or disables server validation.

`SET SSL SERVER VALIDATE NAME [ENABLE|DISABLE]`
Enables or disables server name validation.

`SET SSL SERVER VALIDATE NAME [CURRENT|IS <name>]`
Sets the machine name to use for server validation.

`SET STATUS {TRUE|FALSE}`
Sets the internal flag (see `IF TRUE/FALSE`) to TRUE or FALSE.

`SET STS {ON|OFF}`
Turns the status bar on or off.

`SET TCP <protocol>`
Sets the protocol to be used on the TCP/IP stack. Valid protocols are. `TELNET`, `RLOGIN`, `GWDIWS`, `GWDSA`, `TNVIP`, `TN3270`, `TN5250` and `RAW`.

`SET TCS {OFF|ON|NONE}`
Sets the TCS enable option in the DKU emulator setup menu (inactive, enable or disable).

`SET TERM {TRUE|FALSE}`
Enables (or disables) the termination script (`$$TERM.SCR`). Note that your termination script should execute `SET TERM TRUE` if it wishes to abort termination and still be called if further attempts are made by the user to terminate the emulator.

`SET TOOLBAR BUTTON n image function`
Sets toolbar button number `'n'`. `'image'` is the internal number of the bitmap image to use in the button; these are numbered from 51 and up in the same order as they are presented in the toolbar setup menu. If you have supplied additional images for the toolbar in a GLBITMAP.DLL file (see the toolbar setup help) then you may access these by adding 1000 to the number of the bitmap in your add-in DLL. `'function'` is the internal number of the function to assign to the button; please refer to the online help for a complete table of these.

`SET TOOLBAR HELP n "help text"`
Sets the help text for a button on the toolbar. These are defined in exactly the same way as for buttons in the button bar created with the script `BUTTON` command.

`SET TOOLBAR {ON|OFF}`
Turns the toolbar on or off.

`SET TOOLBAR SIZE n`
Sets the number of buttons in the toolbar.

`SET TRANSFER ANSI`
Resets the Text files in OEM charset option in the Text transfer setup menu.

`SET TRANSFER COMMAND <command>`
Sets the host file transfer command in the file transfer setup menu.

`SET TRANSFER OEM`
Sets the Text files in OEM charset option in the Text transfer setup menu.

`SET TRANSFER OVERWRITE {ON|OFF}`
Sets the file overwrite option in the File transfer setup menu.

`SET TRANSFER PAUSE {ALWAYS|NEVER|FAILED}`
Sets the 'Wait after transfer' option in the File transfer setup menu.

`SET TSM8 {ON|OFF|TSM|TSS}`
Sets the TSM8 enable option in the VIP emulator setup menu (ON, OFF) or alternatively sets TSM8 into TSS or TSM mode (TSM, TSS).

`SET TRCWIN {ON|OFF}`
Enables or disables the debug trace window for script source. See the `TRACE` command for more details.

`SET TTYPE terminal_type`
Sets the terminal type for the telnet, rlogin and TNVIP protocols over TCP/IP. You should restrict your choices to terminal types recognized by the relevant protocol, especially for TNVIP (see the communications setup menu for valid TNVIP types).

`SET UPDATE {ENABLE|DISABLE}`
This enables or disables screen updating. This is a permanent setting that can only be changed with another `SET UPDATE` command, and thus overrides the more usual `SCREEN ON/OFF` command.

`SET UVTI <number>`
Sets the return status for UVTI script execution (values between zero and 255 are permitted). The value is available either as a requestable DDE item or as an immediate DDE advisement (see the *DDE reference* appendix to the *User's Guide*).

`SET WARNINGS {ON|OFF}`
Enables or disables warning messages that might stop unattended scripts, for example, the 'disk full' message.

`SET WALLPAPER filename`
Sets the current screen wallpaper.

`SET WALLPAPER {SCROLL|STATIC|STRETCH|TILE}`
Sets wallpaper attributes.

`SET XFER {ENABLE|DISABLE}`
Enables or disables the ability to perform file transfers.

The SETMACRO command

Syntax: `SETMACRO <number> <string>`

This is used to change the value of one of the keyboard macros. These are numbered from zero to 999. By default, the first ten of these macros are loaded on the ALT+0 to ALT+9 keys. For example:

```
SETMACRO 5 "New macro^M"
```

changes the value loaded onto the ALT+5 key to "`New macro`" and a carriage return character. Macros from 10 to 63 may be accessed either by reconfiguration of the keyboard, the button bar and the tool bar.

The SHELL command

Syntax: `SHELL <command/file>`

This command executes an application, or alternatively the application that is associated with the file type of the specified file. For example:

```
SHELL "myfile.txt"
```

would normally invoke notepad for the `myfile.txt` file, whereas

```
SHELL “notepad”
```

would just execute notepad. See also `ADMSHELL` if you need to invoke the application as an administrator.

The SHOW command

Syntax: `SHOW <string>`

The `SHOW` command will display the string specified on the terminal, but in contrast to `MESSAGE`, will not follow the string with a CRLF sequence. The string will not be sent to the host.

The SNDLINE command

Syntax: `SNDLINE <string>`

The same as the `SEND` command with the modification that a terminator (usually CR or ETX) is also sent at the end of the string. This avoids the need for continual use of the `^M` definition. If you are running in text or forms mode then `SNDLINE` will 'transmit' the text rather than add a carriage return. The text will however be displayed on the screen in that the standard emulation procedures are used for this. If you need to send directly to the communications line without displaying the text, use `TRNLINE`.

The SPEED command

Syntax: `SPEED <speed>`

This is used to set the speed of the communications line. It is only needed if you need to change the speed from the setting that was used when the script was started. For example:

```
SPEED 9600
```

The permitted values are the same as those displayed in the communications setup menu.

The SPLIT command

Syntax: `SPLIT <%var> <%var>`

This command lets you split a string into two parts depending upon the string contents. Given a command:

```
SPLIT %1 %2
```

then `%1` is the string and `%2` is the string to scan for inside `%1`. Note that literal strings are not allowed here. If the string `%2` is NOT contained in `%1` then `OK` is set false and the contents of `%1` and `%2` remain unchanged. If `%1` ***does*** contain `%2` then `OK` is set true, and the part of the string to the ***left*** of `%2` inside `%1` is moved to `%2`, while the part to the ***right*** remains in `%1`. ***Both*** strings are changed, in other words. This is probably best illustrated with an example:

```
ASSIGN %11 "1,2,3"
ASSIGN %12 ","
SPLIT  %11 %12
```

After this operation, variable `%11` will contain `"2,3"` and variable `%12` will contain `"1"`.

The STITLE command

Syntax: `STITLE <string>`

This command is used to change the basic string used in the caption bar for the scrollback window.

The STRACE command

Syntax: `STRACE {ON|OFF}`

This command turns debug mode on or off (same as toggling debug mode in the File menu). Debug information is displayed in a separate window and includes logging of data sent and received as well as other internal information that may of use when debugging problems with the program.

The STRIP command

Syntax: STRIP {YES|NO}

This command may be used to turn stripping of parity from the communications line on and off (see *Communications Setup*).

The SUBRIGHT command

Syntax: SUBRIGHT <%var> <string> <length>

This command allows you to extract the right-hand portion of a string into one of the script variables (see SUBSTR below). <%var> is the number of the variable in which the answer is to be placed, and <length> is the maximum number of characters to extract from the right-hand end of <string>. If the string does not contain that many characters then the answer will just be the entire string. For example, if internal variable number 8 contained the value "ABCDEF" then the command:

```
SUBRIGHT %1 %8 2
```

would place the value "EF" into variable number 1.

The SUBSTR command

Syntax: SUBSTR <%var> <string> <position> <length>

This command allows you to extract part of a string into one of the script variables. <%var> is the variable in which the answer is to be placed, <position> is the position of the first character to extract, and <length> is the maximum number of characters to extract. If the string does not contain that many characters then the answer will be cut short at the end of the string. For example, if internal variable number 8 contained the value "ABCDEF" then the command:

```
SUBSTR %1 %8 2 3
```

would place the value "BCD" into variable number 1.

The SUBTRACT command

Syntax: `SUBTRACT <%var> <number>`

The `SUBTRACT` command allows you to compute the difference between two numbers. The first parameter must be a script variable, while the second may be a script variable or a constant. The result of subtracting the second number from the first is placed in the script variable specified first. For example:

```
SUBTRACT %5 144
```

subtracts 144 from the present contents of the `%5` variable, leaving the result in `%5`. Note that the result may be stored in exponential format to keep maximum precision. If you need to print a result that may be outside the range 0.01 to 32767, you can use the `TRUNCATE` command to format the number in a more suitable way. If the subtraction can be performed correctly then the `OK` variable is set true. If not (because one of the two operands was non-numeric) then it's set false.

The SWITCH command

Syntax: `SWITCH <string>`

The `SWITCH` command marks the start of a switch construct. These are used to test the contents of a script variable (or built-in variable) for one of several alternatives. An example is probably the best way to introduce the `SWITCH` syntax:

```
SWITCH $KEYPRESS
  CASE "1"; MESSAGE "One"
  CASE "2"; MESSAGE "Two"
  CASE "3"; MESSAGE "Three"
  DEFAULT;  MESSAGE "Invalid"
ENDSWITCH
```

What's happening here is that we are using the SWITCH command to introduce a block of statements, terminated with the ENDSWITCH statement. The SWITCH command takes one parameter, which specifies the name of the variable to be tested. It's followed by a series of CASE statements, which specify the alternatives we wish to check on. These are checked one at a time until a match is found, at which point the statements between the matching CASE and the next CASE statement are executed. Only these statements are executed - when the next CASE (or DEFAULT) is found then control goes straight to the statement following the ENDSWITCH statement. The DEFAULT case is used only when none of the preceding CASE statements produce a match, and it must be specified after all the other CASE statements if it's used. Any number of statements may be used for each CASE; if you in the course of the procedure for any particular CASE want to skip immediately to the ENDSWITCH statement then you may use the EXITSWITCH statement for this purpose. Note that unlike the corresponding C language construct, control passes directly to ENDSWITCH as soon as a matching case has been processed; if you want more 'C-like' functionality then use the CSWITCH command instead of SWITCH.

SWITCH commands may be nested up to a maximum of 20 levels; there is no limit on the number of CASE statements for each level.

The TCKEY command

Syntax: `TCKEY <fk_number>`

The TCKEY command applies to the Atlantis V8 interface only, and tells the emulator to simulate the effect of pressing a shifted function key when connected to a TCU/TCS. Some special values apply, besides the function key numbers. A 'reset' is given by key number 13, while in a TM environment you may use 13 for 'operator' and 14 for 'break'.

The TIMEOUT command

Syntax: `TIMEOUT <seconds>`

This tells Glink how long (in seconds) to wait before continuing when a RECEIVE command is looking for a certain string from the host and doesn't find it. The default value is 60 seconds if you don't change it with a command like:

```
TIMEOUT 30
```

Timeouts may be specified to a resolution of 0.1 seconds, in other words the command:

```
TIMEOUT 1.2
```

is legal. The timeout set with this command will also affect the current timeout value for any active `ON TIMEOUT` command.

The TITLE command

Syntax: `TITLE <string>`

This command is used to change the basic string used in the caption bar.

The TRACE command

Syntax: `TRACE [<tenths>]`

The `TRACE` command gives you a separate debugging window which will show you the script that is currently executing and highlight the actual line that is executing. Additionally there is a running display of the line number in the script that is currently executing, shown in the status bar. If you are using nested scripts then the `TRACE` command must be included in each script to be traced. The `TRACE` logic has been written to add as little overhead as possible to the execution of the script, but will of course run slightly more slowly than in the case where `TRACE` has not been used. The `TRACE` command has an optional numeric parameter. If this is present, it specifies a delay in units of tenths of a second. The delay will be made between execution of each script line, and may be found useful for following the path taken by more complex script operations.

If you prefer only to see the line numbers in execution without the additional trace window you can use SET TRCWIN OFF to specify this. Additionally you can use SET TRCWIN ON to enable the window from a called script where an overlying script has removed the window (or the window had earlier been closed manually).

The TRANSMIT command

Syntax: `TRANSMIT <string>`

This command sends data to the host. Unlike `SEND`, no emulation will be done on the local screen for VIP TEXT and FORMS modes; this may be useful for sending data that you don't want to be visible on-screen.

The TRIM command

Syntax: `TRIM {LEFT|RIGHT|BOTH} <%var>`

This command removes leading and/or trailing spaces and control characters from the specified script variable.

The TRNLINE command

Syntax: `TRNLINE <string>`

This command sends a message to the host. Like `SNDLINE`, the appropriate terminator will be added to the message. Unlike `SNDLINE`, no emulation will be done on the local screen for VIP TEXT and FORMS modes; this may be useful for sending commands that you don't want to be visible on-screen.

The TRUNCATE command

Syntax: `TRUNCATE <%var> <decimals>`

The truncate command allows you to format the current contents of a script variable with a given number of decimals. The command is basically designed for use with the arithmetic operations `ADD`, `SUBTRACT`, `MULTIPLY` and `DIVIDE`, which all will provide their output in exponential format (e.g. 1.000000000 E+11) for results outside the range 0.01 to 32767 in order to preserve precision in partial results. The `TRUNCATE` command has two parameters: the script variable to be formatted, and the number of decimals required. For example:

```
TRUNCATE %4 0
```

formats the present contents of `%4` with no decimals. The `OK` variable is set by this command depending upon whether the present contents of `%4` actually are numeric. By default, the `TRUNCATE` command will actually round the number to the closest value; if you actually want truncation then you can get this using the `SET ROUND OFF` command.

The TSMDIR command

Syntax: `TSMDIR <string>`

This command sets the name for both the TSM8 and TCS forms directory. This is the base directory, not necessarily the actual directory used to store the forms, which may be a subdirectory of this specified by the host.

The UCASE command

Syntax: `UCASE <%var>`

This command can be used to convert a variable into upper case (capital letters). It just needs one parameter, the variable to be converted. For example:

```
UCASE %1
```

Note that for the conversion of 'high ASCII' characters to be performed correctly then you must have defined your country code and codepage correctly, or some characters may not be interpreted.

The UNMENU command

Syntax: `UNMENU`

The `UNMENU` command removes the last menu presented on the screen by the `MENU` command. This is an 'expert' command and should be used with care. It is the user's responsibility to ensure that the menu is actually on the screen at the time. If only one menu is active then the `NOMENU` command is to be preferred. This particular command is provided to tackle the case where one menu has been 'tiled' onto another and you wish to make the return to the 'main' menu look more professional. Note that the main menu in this case may not be executed again until it has been removed from the screen or undefined results will occur. The command provides for the case where some kind of 'information' window is desired for the top level.

The UPLOAD command

Syntax: `UPLOAD <string>`

This command allows you to override the predefined upload directory, normally to avoid having to switch to a particular directory containing the files you wish to transmit to the host system. It is good practice to 'save' the current upload directory before you start and restore it again when you are finished, thus:

```
ASSIGN %11 $UPLOAD
UPLOAD "C:\FILES\"
....
UPLOAD %11
```

The URLSHOW command

Syntax: `URLSHOW <urlstring> <title>`

This command invokes your default browser to display the URL specified in `<urlstring>`. The `<title>` parameter is provided for compatibility with Glink/Java, where it's used to specify a particular window in which to display the URL. Note that if you specify the name of a file rather than a URL then Glink will attempt to invoke whatever application has been configured as the default for the file type in question. For example:

```
URLSHOW "http://www.gar.no"
```

See also the SHELL and ADMSHELL commands.

The WELCOME command

Syntax: WELCOME

The WELCOME command takes no parameters and quite simply shows the configured "welcome" menu. This is basically designed for use by the initialization script $$INIT.SCR, which is supplied with the software and does the first-time setup whenever the program is started for a non-existent configuration file.

The WHEN command

Syntax: WHEN <!pattern> <script command>

This command allows you to perform specific actions when one of your predefined patterns is found coming from the line. This allows you to 'program' your script without having to interrupt the normal 'flow'. Here <!pattern> is the pattern number to search for and <script command> is any valid script command. For example, if the host sends the prompt 'More?' at the end of each 'page' of output, and you wish to have your script send a carriage return every time it 'sees' this text then you could use the statements:

```
PATTERN !1 "More?"
WHEN !1 SNDLINE ""
```

It's important to understand that the WHEN statement above does NOT wait for input from the host when it's executed in the script file. It tells Glink what to do while the rest of the script is executing if the specified pattern is received (a script consisting just of the two statements above would terminate immediately, for example). Note that the comparison performed is by default case-sensitive, but that this may be changed using the SET CASE OFF command. If SET IDLE is in effect (and SET IPATTERN is OFF) then the idle delay will be applied before the WHEN statement is executed. Note that this means that there is a possibility that other patterns may be matched (and associated WHEN statements executed) while the script is waiting for the host to become idle.

The $WHEN internal variable always contains the number of the most recently activated WHEN command. You may use this to provide a common routine for several WHEN statements where much of the processing is common to several patterns, testing on which pattern actually was received only when necessary.

The WHILE command

Syntax: WHILE <condition>

This is quite similar to the IF command, with the exception that the command introduces a block of statements that should be executed repeatedly until the condition specified in the WHILE statement fails. The end of the block is signalled using the ENDWHILE command, giving the following structure:

```
WHILE <condition>
  (statements)
ENDWHILE
```

Any condition legal for an IF command may be specified for the WHILE condition; see the IF command on page 92 for details. An example of the WHILE command in use:

```
ASSIGN %1 11
WHILE (%1 LEN 20)
  FRDL #1 _1
  ADD %1 1
ENDWHILE
```

This assumes that a file has been opened for reading as file #1. The first line assigns a value of "11" to variable %1. The WHILE statement tells us that the two statements between it and the ENDWHILE are to be repeated so long as the value of %1 is less than or equal to twenty. The two statements in question are reading a line from the file into the variable whose number is contained in %1 and then incrementing %1 by 1. The effect of this example is therefore to read ten lines from the file into variables %11 to %20 inclusive.

The WINDOW command

Syntax: `WINDOW <option>`

This command gives the script control over the appearance of the main Glink Window. The following are available for `<option>` in the `WINDOW` command:

`HIDE`	Hides the window and passes activation to another window.
`FLASH`	'Flashes' the window once (whether as an icon or a normal window). Repeat the command if necessary.
`MAXI`	Activates the window and displays it as a maximized window.
`MINI`	Minimizes the window and activates the top-level window in the window-manager's list.
`NORMAL`	Activates and displays the window. If the window was minimized or maximized, it will be restored to its original size and position.
`NOTTOP`	Causes the window to revert to normal operation (see `WINDOW TOPMOST`).
`RESTORE`	Same as `WINDOW NORMAL`.
`SHMINI`	Activates the window and displays it as iconic.
`SHMN`	Displays the window as iconic. The window that is currently active remains active.
`SHOW`	Activates the window and displays it in its current size and position.
`SHNA`	Displays the window in its current state. The window that is currently active remains active.
`SHNOACT`	Displays the window in its most recent size and position. The window that is currently active remains active.
`TOPMOST`	Specifies that the emulator window should always remain on top, even when not active.

Be careful with the `WINDOW HIDE` command - if your script hides the window then it is also responsible for showing it again before terminating. Failure to do this may result in Glink becoming inaccessible from the mouse and/or keyboard.

The WKEY command

Syntax: `WKEY`

`WKEY` tells Glink to wait for the user to press any key before continuing. This would typically be used just after presenting some message that you would like the user to read before the script continues executing.

The DBOX command

The DBOX command allows you to define your own dialog boxes directly from a script file, using most of the graphic controls supported by Windows, such as check boxes, buttons, radio buttons and list boxes.

The general format of the DBOX command is as follows:

```
DBOX [MINIMIZE] X Y W H ["Caption"]
  [control definitions]
ENDDBOX ["HelpFile"]
```

If the MINIMIZE option is specified then the dialog box will be equipped with a minimize button.

X and Y specify the initial position of the dialog box (relative to the current position of the Glink window), while W and H specify the width and height. These are all in 'dialog units', a special type of measurement designed to make positions and sizes fairly independent of what size of screen the dialog box is to be displayed on. The optional caption text will be displayed in the caption bar of the dialog box if provided.

If you would rather have the dialog box positioned centrally in the desktop window rather than specify a particular position, then you can replace the X Y specification in the above with the keyword CENTER.

If you don't wish to specify the size of the dialog box yourself, but have it 'fitted' around the dialog box elements you define, then you can replace the W H specification in the above with the keyword AUTO.

The optional help filename (if specified) must be the name of a valid Windows help file (with the full path if necessary). This help file will be called if you press F1 while in the dialog box. You may also define a specific Help button if you wish.

The definition of the dialog box has intentionally been designed to be as equivalent as possible to the way these are defined in other products (for example in Windows SDK resource scripts). This should make it easy to 'import' dialog boxes that you may already have defined elsewhere.

General

You should specify the contents of all variables needed for the dialog box before you use the `DBOX` command. Once the `DBOX` command has been encountered only valid dialog box commands will be accepted by the script compiler. Note that the usual ways of specifying strings as concatenated text inside parentheses may be used freely inside the definition of the dialog box.

Once you have set up all the necessary variables (and in some cases, files) you can define the dialog box with the `DBOX` command. When the `ENDDBOX` command is reached the dialog box will be shown on the screen and can be used with exactly the same keyboard and mouse functionality as any other dialog box in Windows.

When you exit from the dialog box by pressing one of the buttons in the dialog box then the script will continue at the statement after the `ENDDBOX` command. At this point you can inspect the results in the different variables that were used inside the dialog box. You may also inspect the value of the internal `$DBOX` variable, which will contain the number of the button that was pressed to exit the dialog. This will contain 1 if the 'OK' button was pressed or the user exited by pressing enter, 2 if the 'Cancel' button was pressed or the user pressed the Escape key, or the button number if another button was pressed.

If the 'Cancel' button was pressed then all variables used inside the dialog box will contain their initial values. The `$KEYPRESS` variable will contain an Escape character (`"^["`) in this case (whether the exit was performed with the keyboard or the mouse). In other cases `$KEYPRESS` will contain a CR (`"^M"`).

The one exception to the rule that the dialog box will be terminated is the help button, which will call the defined help file (if any).

No actions will be performed by the dialog box other than returning values in the script variables you have specified. So if you are using the dialog box to select a file name, for example, any actions you wish to perform on the file must be performed by the script itself after the dialog box has been executed.

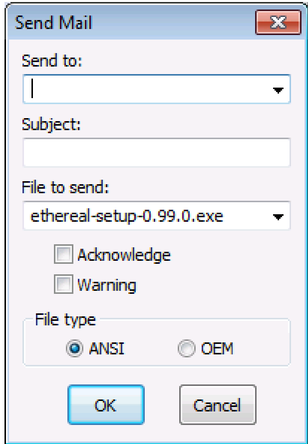
If you have specified a new font for menus using the `MFONT` command then this font will also be used for your own dialog boxes. The requirements for the naming of the font are slightly more specific; but if the named font cannot be used the dialog box will still be executed using the system font. Trial and error will show you which `MFONT` statements are acceptable and which are not.

Here is an example of a script fragment using a dialog box to collect information about a message to be sent into an online system, showing the use of most of the different control types in practice. You may not understand all of the commands used here immediately, but this gives an overview of what an actual dialog box definition will look like in practice:

```
assi %1 "@ADDRESS.LST"
assi %2 ""
assi %3 "@F \TEMP\*.*"
assi %4 0
assi %5 0
assi %6 1
mfont "Tahoma" 8
dbox 10 10 132 164 "Send Mail"
  ltext            6   4 120  10 "&Send to:"
  combobox         6  14 120  90 %1
  ltext            6  30 120  10 "S&ubject:"
  edittext         6  40 120  12 %2 hscroll
  ltext            6  56 120  10 "&File to send:"
  combobox         6  66 120  90 %3
  checkbox        20  82 120  12 "&Acknowledge" %4
  checkbox        20  94 120  12 "&Warning" %5
  groupbox         6 110 120  24 "&File type"
    radiobutton   25 120  50  12 "ANSI" %6 1
    radiobutton   75 120  50  12 "OEM" %6 2
  endgroup
  defpushbutton   25 140  36  18 "OK" 1
  pushbutton      75 140  36  18 "Cancel" 2
enddbox
```

This dialog box starts with a heading text and a combo box showing a list of names, which are picked up from a local file and from which you can select the name to use. After this we have an edit control (again with a heading) into which the subject of the message can be entered. An option specifies that the text should scroll horizontally if there is not room in the window shown on the screen. After this, we have two checkboxes where you can turn on acknowledgement and warning options. This is followed by a group with two radio buttons where the file type can be selected. Finally, we have the familiar 'OK' and 'Cancel' buttons at the bottom of the dialog box.

The actual appearance of the dialog box on the screen may be found helpful in making this clearer:



Dialog units

All positions and sizes used in a dialog box definition use 'dialog units'. One dialog unit in the horizontal direction is set to one quarter of the average character width of the font being used and one vertical unit to one eighth of the average character height. This is the standard way of defining dialog box positions and sizes and helps avoid the problems involved in defining a dialog box which will turn out approximately the same on screens of widely differing resolutions and sizes.

A 'normal' height for a single line control like a checkbox or radio button is 12 dialog units. The width of such a control depends of course on what text is going to be displayed. However, you can see from the above definition that somewhat more than four times the number of characters you are going to use will be roughly what is needed. In general, use more space than you think you will need rather than less.

Dialog box controls

The following commands are used to define dialog box controls. These are the only commands permitted to appear between the `DBOX` and `ENDDBOX` commands, and are not valid anywhere else in a Glink script. Indirection of variables is not permitted in those places where Glink variables are referenced.

Command summary

```
AUTOGROUP                      "Text"                   [options]
BBUTTON          X  Y  W  H    "Text"         number    [options]
BITMAP           X  Y  W  H    "Text"
CHECKBOX         X  Y  W  H    "Text"  %N               [options]
COMBOBOX         X  Y  W  H            %N               [options]
CTEXT            X  Y  W  H    "Text"                   [options]
DEFPUSHBUTTON    X  Y  W  H    "Text"         number    [options]
EDITTEXT         X  Y  W  H            %N               [options]
ENDGROUP
ENDHGROUP
ENDVGROUP
GROUPBOX         X  Y  W  H    "Text"                   [options]
HGROUP           X  Y  W  H
IBUTTON          X  Y  W  H    "Text"         number    [options]
ICON             X  Y  W  H    "Text"
LISTBOX          X  Y  W  H            %N               [options]
LTEXT            X  Y  W  H    "Text"                   [options]
PUSHBUTTON       X  Y  W  H    "Text"         number    [options]
RADIOBUTTON      X  Y  W  H    "Text"  %N     number    [options]
RTEXT            X  Y  W  H    "Text"                   [options]
SIZEBUTTON       X  Y  W  H    "Text"   DW DH           [options]
TRACKBAR         X  Y  W  H            %N LV HV         [options]
VGROUP           X  Y  W  H
```

`X Y` defines the position (in dialog units) relative to the dialog box itself, where "0 0" is the top left corner of the box (not including the caption or the frame). `W H` defines the width and height of the control (again in dialog units). Any or all of these may be specified using script variables rather than constants.

NOTE: if you are defining a horizontal or vertical group using the `HGROUP` or `VGROUP` command then only the width or the height will be defined with each control in the group.

In place of `X Y` you may also use the keyword `DOWN` or `RIGHT`. If you use one of these the control will be placed immediately below (or immediately to the right of) the previous control. Additionally you may specify a positive or negative offset from this, for example:

```
PUSHBUTTON 30 120 36 24 "OK" OK
PUSHBUTTON RIGHT+24 36 24 "Cancel" Cancel
```

In the above you must **NOT** have any spaces in the string `"RIGHT+24"`.

In place of `W H` you may also use the keyword `SAME`. This will produce a control with exactly the same size as the previous defined control.

Additionally, if you require to use only one of the coordinates from the previous control, you can use `OLD` or `NEW` in place of either of the `X` or `Y` positions, and `OLD` in place of either of the `W` or `H` specifications. In the case of the `X` and `Y` positions, `OLD` refers to the left/top position of the previous control, while `NEW` refers to the right/bottom of the previous control. Here again you may also specify optional positive or negative offsets. This means that `DOWN` is the same thing as `OLD NEW`, while `RIGHT` is the same thing as `NEW OLD`. For example, you could write:

```
EDITTEXT           10     10  100  12 %1
DEFPUSHBUTTON OLD+10 NEW+5    30   20 "OK" OK
PUSHBUTTON    NEW+20   OLD SAME "Cancel" Cancel
```

`"Text"` is the text to be displayed in the control (edit controls and list boxes take their text from the associated variable). This may also be defined in any of the normal formats used by the script language, for example a valid `LTEXT` statement could be:

```
LTEXT 6 12 90 12 ("Value of " %15)
```

In all cases where a text is displayed with a control, you can add an ampersand (`&`) before the letter you wish to use as a 'hot key' to move directly to that item from the keyboard. For example, you could define:

```
CHECKBOX 6 12 120 12 "&Collect output reports" %5
```

`%N` defines the associated variable, used to define both the initial state and to return the final state of the control. More information on this will be given when the different controls are defined.

`'Number'` defines the number of the pushbutton (or default pushbutton or `IBUTTON`). Buttons may be numbered from 1 to 63, and will always result in the dialog box being terminated when pressed. The exception is the help button; see the definition of the `PUSHBUTTON` statement on page 205 for more information.

By convention the OK button should be numbered 1 and the Cancel button (if any) should be numbered 2. This will preserve the normal functionality for the ENTER and ESCAPE keys (you may use the keywords `OK` and `CANCEL` for the button number instead of specifying the actual number). The value of the button used to terminate the dialog box will be returned in the internal `$DBOX` variable.

`[options]` is a list of specific options which apply to the control. The following options are supported (not all will apply to every control type):

`ALPHABETIC (COMBOBOX, EDITTEXT)`
The edit control will only accept alphabetic characters.

`[attributes] (most)`
Attributes may be set on most types of text field (see specific items for applicability, and note that buttons are a special case where attributes may not be used because of Windows limitations). Attributes that may be set include:

```
[LIGHT] RED | GREEN | BLUE | YELLOW | CYAN |
        MAGENTA | GREY | GRAY | WHITE
BOLD
UNDERLINE
```

`AUTOTAB (COMBOBOX, EDITTEXT)`
Must be combined with the `MAXLENGTH` option and specifies that an automatic tab to the next field be performed when the maximum length of the field is reached.

`BOTH (TRACKBAR)`
The trackbar should display ticks on both sides of the slider.

`BOTTOM (TRACKBAR)`
The trackbar should display ticks underneath the (horizontal) trackbar. This is the default for horizontal trackbars.

`CENTER (EDITTEXT)`
Displays centered text. Applies to multiline edit controls only, but you can get centering by using a multiline control with no scroll bar and only enough space for a single line.

`DIGIT (COMBOBOX, EDITTEXT)`
The edit control will only accept digits.

`DISABLED (all)`
Disables the control so that input may not be entered.

`DRDW (COMBOBOX)`
Defines a 'dropdown' combo box (one that has a 'hidden' list box which drops down when you press the icon next to the edit field).

`DRLS (COMBOBOX)`
Defines a 'dropdown list' combo box (the same as the 'dropdown' style except that you cannot type into the text field).

`FILL (COMBOBOX, EDITTEXT)`
The edit control must be filled to the maximum defined length (as defined by the `MAXLENGTH` option) if any data is entered.

`FOCUS (all)`
Sets the input focus to this control initially.

`GROUP (all)`
Defines the start of a group.

`HBAR (COMBOBOX, LISTBOX, EDITTEXT)`
Provides a horizontal scrollbar for scrolling of the text. For `EDITTEXT`, sets the `HSCROLL` and `MULTILINE` options automatically, in that these are the only types of control for which the scroll bar is useful.

`HSCROLL (EDITTEXT)`
Allows horizontal scrolling of the text.

`INACTIVITY n (all BUTTONs)`
If there has been no activity in the dialog box for the specified number of seconds then this button will be pressed automatically. Use the `TIMEOUT` option if you want the button to be pressed with a timeout that starts when the dialog box is first shown.

`LEFT (CHECKBOX, RADIOBUTTON, TRACKBAR)`
Displays the text (or the trackbar ticks) to the left of the button.

`LIST (COMBOBOX, LISTBOX)`
Specifies that an inline list of contents will follow this listbox definition.

`LOWERCASE (EDITTEXT)`
All text entered is converted to lower case.

`MAXLENGTH n  (COMBOBOX, EDITTEXT)`
The edit control will accept a maximum of n characters.

`MINLENGTH n  (COMBOBOX, EDITTEXT)`
The edit control must contain a minimum of n characters.

`MULTILINE (EDITTEXT)`
Multiple lines of text may be entered.

`NOGROUP (all)`
This control is not the start of a group.

`NOSIZE (BBUTTON, BITMAP)`
Suppresses resizing of bitmaps.

`NOTABSTOP (all)`
This control is not a tab stop.

`NOTICKS (TRACKBAR)`
The trackbar should not display any ticks.

`NUMERIC (COMBOBOX, EDITTEXT)`
The edit control will only accept numerics (0-9, comma, period, +, -).

`PASSWORD (EDITTEXT)`
Text entered should not be displayed on the screen.

`RANGE min max (EDITTEXT)`
Specifies a minimum and maximum value for the entry (requires that the entry also be defined as `NUMERIC`).

`READONLY (EDITTEXT)`
You are not allowed to modify the text (but can for example mark it and copy it to the clipboard).

`REQUIRED (COMBOBOX, EDITTEXT)`
You must enter at least one non-space character in this field.

`RETEDIT (COMBOBOX)`
Specifies that the contents of the edit box in a dropdown combo box should be returned (normally the currently selected item in the list itself will be returned irrespective of the contents of the edit control).

`REVERSE (COMBOBOX)`
Display the entries in the list box using only the field after the ETX character in the data for the item (see the description of the `COMBOBOX` for more details).

`RIGHT (EDITTEXT, TRACKBAR)`
The text should be displayed with right justification. Applies to multiline edit controls only, but you can get right justification by using a multiline control with no scroll bar and only enough space for a single line. For trackbar controls, this specifies that a vertical trackbar should display its tick marks on the right.

`SIMPLE (COMBOBOX)`
Defines a 'simple' combo box (this has an edit control and a permanently displayed list box).

`SORT (LISTBOX, COMBOBOX)`
The items in the box should be sorted before display.

`TABPOSITION (p1 p2 ...) (LISTBOX, COMBOBOX)`
Specifies tab positions to be used in the list box (in dialog units). Up to eight positions may be specified. If more tabs are used in the text of the item being displayed than are specified with the `TABPOSITION` definition then the distance between the last two positions specified will be used to generate additional tab stops. For example:

```
TABPOSITION (12 32)
```

This will set tab stops at 12, 32, 52, 72 and so on. In the case of the combo box, tabs in the text item (defined with `^I`) will simply be replaced by spaces when the item is displayed. Default tab positions are at 20, 40, 60, 80 and so on.

`TABSTOP (all)`
This item has a tab stop.

`TIMEOUT n (all BUTTONs)`
If the dialog box has not been exited after the specified number of seconds then this button will be pressed automatically. Use the `INACTIVITY` option if you want the button to be pressed with a timeout that is restarted whenever there is activity in the dialog box.

`TOP (TRACKBAR)`
The (horizontal) trackbar should display its tick marks above the slider rather than below.

`UPDOWN (EDITTEXT)`
Add an updown (spinner) control to this edit field. Should normally only be used on numeric fields.

`UPPERCASE (EDITTEXT)`
Text should be converted to upper case as entered.

`VBAR (EDITTEXT)`
Provides a vertical scrollbar for scrolling of the text. Sets the `VSCROLL` and `MULTILINE` options automatically, in that these are the only types of control for which the scroll bar is useful.

`VERTICAL (TRACKBAR)`
The trackbar control should be displayed vertically rather than horizontally.

`VSCROLL (EDITTEXT)`
The (multiline) text should be allowed to scroll vertically.

`WRAP n (EDITTEXT)`
The text output to a file from a multiline edit control will be word-wrapped at column n. Note that although text entered continuously into such a control will be wrapped on the screen, it will not be wrapped on the file unless the `WRAP` option has been specified.

Most of these options will only affect aspects of the control they are used with. However, the `GROUP` and `TABSTOP` (also the `NOGROUP` and `NOTABSTOP`) options will affect the behaviour of the dialog box as a whole.

Dialog box elements

Automatic group boxes

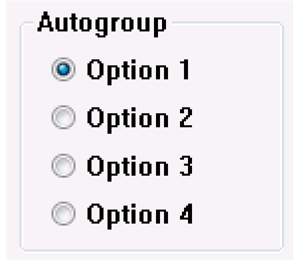
Syntax:

```
AUTOGROUP "Text" [options]
```

Definition:

This defines an 'automatic' group box. This is exactly the same as a group box, but you need not specify the position or dimensions of the box. It will automatically be adjusted so as to enclose the following items in the group.

The specified text will be displayed in the top border of the enclosing box.



Available options:

[attributes], DISABLED, FOCUS, GROUP, NOGROUP, NOTABSTOP, TABSTOP

Default options:

GROUP

Bitmap buttons

Syntax:

```
BBUTTON X Y W H "Text" number [options]
```

Definition:

This is exactly the same as the normal PUSHBUTTON command, except that instead of using a text inside the button it uses a bitmap to represent the action corresponding with pressing the button (in much the same way as the toolbar). The bitmap used may be a predefined bitmap delivered with the program, a bitmap extracted from a library (.DLL) file, or a bitmap contained in a bitmap file (.BMP). The predefined bitmaps delivered with the program are the same as those you see in the toolbar setup menu. They are numbered from 51 and up in the same order as they are presented in the list box in that menu. To use one of these, specify the number of the bitmap with a '#' in front. To load a bitmap from a BMP file such as those produced by Paintbrush, just specify the file name. To load a bitmap from a DLL library, specify the name of the library and then the name of the bitmap resource inside the library.

The bitmap will be resized so as to fit the space provided for it. If you would prefer this not to happen, then specify the NOSIZE option in the bitmap definition. In that case the bitmap will be centered and/or clipped rather than resized.

Examples:

```
BBUTTON 60 20 31 27 "#93" OK
BBUTTON 60 20 50 35 "C:\WORK\BITMAP.BMP" OK
BBUTTON 60 20 50 35 "C:\BITMAPS\MYLIB.DLL BIT_5" 5
```

Available options:

DISABLED, FOCUS, GROUP, INACTIVITY, NOGROUP, NOSIZE, NOTABSTOP, TABSTOP, TIMEOUT

Default options:

TABSTOP

Bitmaps

Syntax:

```
BITMAP X Y W H "Text"
```

Definition:

This allows you to place a bitmap inside a dialog box. The bitmap will not be active in any way (see the preceding BBUTTON command for a way to use a bitmap inside a pushbutton). The bitmap used may be a predefined bitmap delivered with the program, a bitmap extracted from a library (.DLL) file, or a bitmap contained in a bitmap file (.BMP). The predefined bitmaps delivered with the program are the same as those you see in the toolbar setup menu. They are numbered from 51 and up in the same order as they are presented in the list box in that menu. To use one of these, specify the number of the bitmap with a '#' in front. To load a bitmap from a BMP file such as those produced by Paintbrush, just specify the file name. To load a bitmap from a DLL library, specify the name of the library and then the name of the bitmap resource inside the library.

The bitmap will be resized so as to fit the space provided for it. If you would prefer this not to happen, then specify the NOSIZE option in the bitmap definition. In that case the bitmap will be centered and/or clipped rather than resized.

Examples:

```
BITMAP 60 20 31 27 "#93"
BITMAP 60 20 50 35 "C:\WORK\BITMAP.BMP"
BITMAP 60 20 50 35 "C:\BITMAPS\MYLIB.DLL BIT_5"
```

Available options:

NOSIZE

Check boxes

Syntax:

```
CHECKBOX X Y W H "Text" %N [options]
```

Definition:

This defines a check box control, which creates a small rectangle which can be either selected or not. The associated text is displayed next to the check box. Each check box operates independently of other check boxes and returns a separate value (compare with radio buttons) but you may wish to group several of them inside a group box if they are related.

`%N` defines the name of the variable you are going to use in order to set the initial state and read the final state when you exit from the dialog box. A value of "0" means unchecked and a value of "1" means checked.

Available options:

`[attributes]`, `DISABLED`, `FOCUS`, `LEFT`, `GROUP`, `INACTIVITY`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`, `TIMEOUT`

Default options:

`TABSTOP`

Combo boxes

Syntax:

```
COMBOBOX X Y W H %N [options]
```

Definition:

This defines a combination box, which contains both a text field and a list box. The current selection from the list box is displayed in the text field. Three variants of the combo box are available and selectable by options:

Simple (SIMPLE)	the list box is always displayed.
Dropdown (DRDW)	the list box is hidden, but opened when you click the icon next to the text field.
Dropdown List (DRLS)	the same as Dropdown, but the edit field is replaced with a fixed text that cannot be edited (but which still shows the value of the current selection)

The associated script variable defines both the initial values to load into the list box, and returns the value actually selected. See the LISTBOX definition on page 201 for more details of how the list box may be loaded. In the case where the dialog box is terminated because the user double-clicked an item from the list in a simple combo box, then the number of the associated variable will be returned in the $DBLCLICK internal variable. In this case the value of $DBOX will be 1 in that the default meaning assigned to a double click is the same as for clicking on 'OK'.

Note that the height H for a dropdown box should specify the height of the box when the list is displayed, not the height when the list is closed.

One additional option relates to the loading of the list box, specified with REVERSE. In this case, you will have specified items for the list box with an ETX separator. Normally this will result in only the first item (before the ETX) being displayed in the edit control and the list box. When the REVERSE option has been specified, the part before the ETX will be displayed in the edit box as usual, but in the list box the part after the ETX will be displayed. This allows you to implement combo boxes of the type where only a system 'code' need be entered in the edit control, but more explanatory items entered into the list box. For example, the list of states shown in the list box example could have been supplied in the form "AL^CAlabama" and so on. The list box would then show the name of the state but the edit control would operate with the postal code.

Available options:

`[attributes]`, `ALPHABETIC`, `AUTOTAB`, `DIGIT`, `DISABLED`, `DRDW`, `DRLS`, `FILL`, `FOCUS`, `GROUP`, `HBAR`, `LIST`, `MAXLENGTH`, `MINLENGTH`, `NOGROUP`, `NOTABSTOP`, `NUMERIC`, `REQUIRED`, `RETEDIT`, `REVERSE`, `SIMPLE`, `SORT`, `TABPOSITION`, `TABSTOP`

Default options:

`DRDW`, `TABSTOP`

Centered text

Syntax:

```
CTEXT X Y W H "Text" [options]
```

Definition:

This defines a centered text control. It creates a simple rectangle that displays the given text centered inside the specified area. The text will wrap around if it would extend past the end of the line (and if you have provided enough room for more than one line).

Available options:

`DISABLED`, `FOCUS`, `GROUP`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`

Default options:

`GROUP`

Default pushbuttons

Syntax:

```
DEFPUSHBUTTON X Y W H "Text" number [options]
```

Definition:

This defines exactly the same as the PUSHBUTTON statement, except that the button is drawn with a thick border, showing you that this is the default response. In most cases this will also be the action taken if you press the enter key. See the PUSHBUTTON statement on page 205 for more details.

Available options:

DISABLED, FOCUS, GROUP, INACTIVITY, NOGROUP, NOTABSTOP, TABSTOP, TIMEOUT

Default options:

TABSTOP

Edit text

Syntax:

```
EDITTEXT X Y W H %N [options]
```

Definition:

This defines a rectangular region into which you can type text. The contents of the script variable specified will be used to set the initial contents of the text field, and will also contain the modified text when the dialog box terminates. For editing of larger amounts of text than can be held in a normal script variable, you may use a file. To do this use the format

```
"@FILENAME.EXT"
```

in the initial contents of the script variable. If you do this then the specified file will be presented in the edit control and the edited results written back to the same file. If the file does not exist then the edit control will be empty initially and the file will be created when the dialog is terminated. If you wish the edit control to initially be empty, but still output to a file, then specify the file name with an additional `"@"` character as follows:

```
"@@FILENAME.EXT"
```

Edit controls which use files for passing data should normally specify the `MULTILINE` option. Note that there is a limit to the amount of text that can be processed inside an edit control (about 50K). We recommend that you reserve use of file-based edit controls for files that are smaller than this.

Available options:

`[attributes]`, `ALPHABETIC`, `AUTOTAB`, `CENTER`, `DIGIT`, `DISABLED`, `FILL`, `FOCUS`, `GROUP`, `HBAR`, `HSCROLL`, `LOWERCASE`, `MAXLENGTH`, `MINLENGTH`, `MULTILINE`, `NOGROUP`, `NOTABSTOP`, `NUMERIC`, `PASSWORD`, `READONLY`, `REQUIRED`, `RIGHT`, `TABSTOP`, `UPPERCASE`, `VBAR`, `VSCROLL`, `WRAP`

Default options:

`TABSTOP`

End of group marker

Syntax:

```
ENDGROUP
```

Definition:

This is simply a marker for the end of the current group of controls. Windows will automatically terminate a group whenever a new control with the `GROUP` option is defined. However, you may find it more convenient to use a specific command for the end of the group, in terms of both readability and functionality.

Available options:

None.

End of horizontal group

Syntax:

```
ENDHGROUP
```

Definition:

This marks the end of a horizontally spaced group of controls.

Available options:

None.

End of vertical group

Syntax:

```
ENDVGROUP
```

Definition:

This marks the end of a vertically spaced group of controls.

Available options:

None.

Group boxes

Syntax:

```
GROUPBOX X Y W H "Text" [options]
```

Definition:

This defines a box that groups together several other items. The specified text will be displayed in the top left corner of the box.

The commands to be grouped must follow immediately after the GROUPBOX definition. Note that there is no built-in mechanism or ensuring that the group box actually physically encloses the group. The definition of the group is that all controls following the group box that do not have the GROUP option set will be included in the group. You may find that the AUTOGROUP command is easier to use in that this does the necessary calculations for you automatically.

Available options:

[attributes], DISABLED, FOCUS, GROUP, NOGROUP, NOTABSTOP, TABSTOP

Default options:

GROUP

Horizontal group

Syntax:

```
HGROUP X Y W H
```

Definition:

This defines the start of a group of controls that will be spaced horizontally for you, without the need for calculation of the position of each control. The HGROUP command specifies the total area that will be available for all items in the group. For each item in the group, only the item width should be specified. The vertical position and the height of each item will be taken from the HGROUP definition, and the horizontal position will be calculated for you so as to provide equal space between each item in the group. HGROUP groups should be terminated with an ENDHGROUP command.

Example:

```
HGROUP         30 70 120 24
  PUSHBUTTON   36   "OK"     OK
  PUSHBUTTON SAME   "Cancel" Cancel
  PUSHBUTTON SAME   "Help"   Help
ENDHGROUP
```

Available options:

None.

Icon buttons

Syntax:

```
IBUTTON X Y W H "Text" number [options]
```

Definition:

This is exactly the same as the normal PUSHBUTTON command, except that instead of using a text inside the button it uses an icon to represent the action corresponding with pressing the button (in much the same way as the toolbar). The icon used may be a predefined icon delivered with the program, an icon extracted from a program file (.EXE or .DLL), or an icon contained in an icon file (.ICO). The predefined icons delivered with the program are the following (see the ICON command for pictures of the icons themselves):

#CANCEL, #EXCLAMATION, #HELP, #INFORMATION, #NO,
#QUESTION, #STOP, #YES

To use an icon from an executable or icon file, simply specify the file name and (in the case of a file containing multiple icons) the number of the icon inside the file. If no number is specified then the first icon (icon zero) will be used.

Examples:

```
IBUTTON 60 20 30 24 "#YES" OK
IBUTTON 60 20 30 24 "\WINDOWS\MORICONS.DLL 4" 5
```

Available options:

DISABLED, FOCUS, GROUP, INACTIVITY, NOGROUP, NOTABSTOP,
TABSTOP, TIMEOUT

Default options:

TABSTOP

Icons

Syntax:

```
ICON X Y W H "Text"
```

Definition:

This allows you to place an icon inside a dialog box. The icon will not be active in any way (see the preceding `IBUTTON` command for a way to use an icon inside a pushbutton). The icon used may be a predefined icon delivered with the program, an icon extracted from a program file (`.EXE` or `.DLL`), or an icon contained in an icon file (`.ICO`). The predefined icons delivered with the program are the following:

To use an icon from an executable or icon file, simply specify the file name and (in the case of a file containing multiple icons) the number of the icon inside the file. If no number is specified then the first icon (icon zero) will be used.

Examples:

```
ICON 60 120 24 24 "#YES"
ICON 60 120 24 24 "C:\WINDOWS\MORICONS.DLL 4"
```

Available options:

None

List boxes

Syntax:

```
LISTBOX X Y W H %N [options]
```

Definition:

This defines a rectangle containing a list of items from which you can make selections (for dropdown list boxes, see the COMBOBOX command on page 192).

The initial contents of the list box are specified by the contents of the %N script variable when the dialog is started, and can be one of the following:

Comma-separated list

This is a simple list of the strings to be included, separated by commas. For example:

```
"Apples,Bananas,Oranges,Pears"
```

would give exactly these four items in the list box. You can specify which of these is to be the initial selection by prefixing it with an exclamation point, for example:

```
"Apples,Bananas,!Oranges,Pears"
```

Note the 80-character limit for contents of script variables, which limits the usefulness of this way of specifying the contents of the list box other than for quite short lists.

You may specify the contents of an item as two elements separated by an ETX character (which you enter as ^C). In this case, only the contents of the field before the ETX character will actually be displayed in the list box. On the other hand, the returned value to your script will consist of the complete contents of the field, both before and after the ETX character. You can use this feature to include codes that you need for the script but wish to hide from the user.

Inline list

For longer lists, you can include a list directly in the script. Do this by specifying the `LIST` option in the definition of the list (or combo) box. Starting in the next line of your script include the desired items in the list box. You can include several items per line with comma separators or one item per line as you please. Leading spaces will be stripped (this allows you to indent the list to make your script more readable). The only requirement is that you should terminate the list with an

```
ENDLIST
```

command. The same convention as described above (initial exclamation point defines the initially selected item) applies here. For example:

```
LISTBOX 12 12 120 150 %1 LIST
  Alabama, Alaska, Arizona, Arkansas, California,
  Colorado, Connecticut, Delaware, Florida,
  Georgia, Hawaii, Idaho, Illinois, Indiana,
  Iowa, Kansas, Kentucky, Louisiana, Maine,
  Maryland, Massachusetts, Michigan, Minnesota,
  Mississippi, Missouri, Montana, Nebraska,
  Nevada, New Hampshire, New Jersey, New Mexico,
  New York, North Carolina, North Dakota, Ohio,
  Oklahoma, Oregon, Pennsylvania, Rhode Island,
  South Carolina, South Dakota, Tennessee, Texas,
  Utah, Vermont, Virginia, Washington,
  West Virginia, Wisconsin, Wyoming
ENDLIST
```

ETX characters may be inserted in the same way as for the comma-based list described above; these should be inserted using the `^C` character in the same way.

If you use an inline list to fill the list box then the initial value of the associated script variable will be ignored.

File-based list

You can provide longer lists by putting them on a text file, one line per item. You place the name of the file into the script variable prefixed with the commercial at (`@`) character. For example you could use `"@LISTBOX.LST"`. One of the lines in the text file may be prefixed with the "!" character in the same way as for comma-based lists. Entries in the file may also include ETX characters as for the other forms.

Directory list

For those cases where you wish to select a filename, you can ask for the list box to be filled with filenames chosen from a specified directory (using wildcards if you wish). Do this by specifying `"@F filespec"` where filespec is the pattern to match. For example `"@F E:\GLINK*.*"` would give a list of all files in the `E:\GLINK` directory.

If you wish the listing to include directories and drives as well as files, use `"@D filespec"` instead. If you wish it to contain only drives and directories use `"@C filespec"`.

In all of the above cases, the specified variable will be returned with the value of the item selected at the time the dialog box was terminated. In the case where the dialog box is terminated because the user double-clicked an item from a list box, then the number of the associated variable will be returned in the $DBLCLICK internal variable. In this case the value of $DBOX will be 1 in that the default meaning assigned to a double click is the same as for clicking on 'OK'.

Available options:

`DISABLED`, `FOCUS`, `GROUP`, `HBAR`, `LIST`, `NOGROUP`, `NOTABSTOP`, `SORT`, `TABPOSITION`, `TABSTOP`

Default options:

`TABSTOP`

Left justified text

Syntax:

```
LTEXT X Y W H "Text" [options]
```

Definition:

This defines a flush-left text control. It creates a simple rectangle that displays the given text left justified inside the specified area. The text will wrap around if it would extend past the end of the line (and if you have provided enough room for more than one line).

Available options:

`[attributes]`, `DISABLED`, `FOCUS`, `GROUP`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`

Default options:

`GROUP`

Pushbuttons

Syntax:

```
PUSHBUTTON X Y W H "Text" number [options]
```

Definition:

This defines a button that you can press. With one exception, pressing such a button will terminate the dialog box and you can then check the values returned from the various fields in the dialog box. The button will be labelled with the text you specify in the definition.

There are three predefined values (which you specify in the 'number' field) for buttons:

`OK` (actual value 1)
This is the normal way to exit from a dialog box.

`CANCEL` (value 2)
This is the normal way to cancel a dialog box. In this case, all values set in script variables before the dialog box was started will be returned to their initial values.

`HELP` (no equivalent value)
This will call the help file for the dialog (which must be specified in the `ENDDBOX` command at the end of the dialog box definition). Pressing `F1` will also call the help file, but providing a help button makes it more obvious that there is help available. This is the exception to the rule that pressing a button will terminate the dialog.

Available options:

`DISABLED`, `FOCUS`, `GROUP`, `INACTIVITY`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`, `TIMEOUT`

Default options:

`TABSTOP`

Radio buttons

Syntax:

```
RADIOBUTTON X Y W H "Text" %N number [options]
```

Definition:

This defines a button, usually part of a group of buttons, where selecting one button automatically deselects all the others in the same group. These are used in situations where only one of a number of options may be selected. The text specified in the command will be shown next to the button.

Normally you will define a group of radio buttons with the same script variable for each button, but with different values for 'number'. Before the DBOX command is executed, you should assign the number of the button you wish to have as the initial selection to this script variable. When the dialog box is terminated the variable will contain the number of the radio button that actually was chosen. For example:

```
ASSIGN %7 "3"
...
DBOX 20 20 150 200 "Example"
...
  AUTOGROUP "Buttons"
    RADIOBUTTON 12 30 120 12  "First choice"   %7 1
    RADIOBUTTON DOWN  SAME    "Second choice"  %7 2
    RADIOBUTTON DOWN  SAME    "Third choice"   %7 3
    RADIOBUTTON DOWN  SAME    "Fourth choice"  %7 4
    RADIOBUTTON DOWN  SAME    "Fifth choice"   %7 5
  ENDGROUP
...
ENDDBOX
```

In this example, the third button will be selected initially. If you selected the fourth choice instead while the dialog box was on the screen then %7 will now contain "4".

Other ways of using variables and radio buttons are possible, so a brief technical discussion of what is actually happening here is in order. When the dialog box is started, each radio button is inspected. If the number specified for the radio button matches the contents of the script variable then the button will be selected; otherwise it will be left unchecked. When the OK button is pressed each radio button is inspected again. First all associated variables are zeroed, and after that every button that is selected will move the associated number into the defined variable. This means that you don't necessarily have to use the same variable for each radio button in the group if that makes life easier in the rest of your script. Normally however you'll find that doing things as shown in the example is the simplest way of handling radio buttons.

Available options:

`[attributes]`, `DISABLED`, `FOCUS`, `INACTIVITY`, `LEFT`, `GROUP`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`, `TIMEOUT`

Default options:

`TABSTOP`

Right justified text

Syntax:

```
RTEXT X Y W H "Text" [options]
```

Definition:

This defines a flush-right text control. It creates a simple rectangle that displays the given text right justified inside the specified area. The text will wrap around if it would extend past the end of the line (and if you have provided enough room for more than one line).

Available options:

`[attributes]`, `DISABLED`, `FOCUS`, `GROUP`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`

Default options:

`GROUP`

Size buttons

Syntax:

```
SIZEBUTTON X Y W H "Text" dw dh [options]
```

Definition:

This defines a button that you can press to change the size of the dialog box interactively. The dw and dh values define the new width and height of the dialog box in dialog units. The button will be labelled with the text you specify in the definition.

When you use this type of button, you will also be defining some of the elements of the dialog box outside the area specified in the `DBOX` command. Note that this also implies that you are not able to use the `AUTO` option for the dialog box size. These elements will not be accessible until the size button is pressed, which will open a larger area of the box and reveal the additional items. In the same way, you are also at liberty to define a button that 'shrinks' the dialog box back to a smaller size.

Example:

```
DBOX 10 10 100 48 "Size example"
  EDITTEXT    10 10 20 14 %1
  SIZEBUTTON  20 30 50 14 "Advanced" 100 88
  EDITTEXT    10 50 20 14 %2
  SIZEBUTTON  20 70 50 14 "Simple" 100 48
ENDDBOX
```

Available options:

`DISABLED`, `FOCUS`, `GROUP`, `NOGROUP`, `NOTABSTOP`, `TABSTOP`

Default options:

`TABSTOP`

Trackbars

Syntax:

```
TRACKBAR X Y W H %N min max [options]
```

Definition:

This defines a 'trackbar', which is displayed as a slider control and has a minimum and maximum value to which it can be set. The initial value is taken from the `%N` variable and the value to which the trackbar is set is returned in the same place. By default, a trackbar will be displayed as a horizontal slider with tick marks underneath the slider. You may use the available options to suppress the tick marks, to make the slider vertical, or to display the tick marks either on the other side or on both sides.

Available options:

`BOTH`, `BOTTOM`, `DISABLED`, `FOCUS`, `GROUP`, `LEFT`, `NOGROUP`, `NOTABSTOP`, `NOTICKS`, `RIGHT`, `TABSTOP`, `TOP`, `VERTICAL`

Default options:

`TABSTOP`

Vertical group

Syntax:

```
VGROUP X Y W H
```

Definition:

This defines the start of a group of controls that will be spaced vertically for you, without the need for calculation of the position of each control. The `VGROUP` command specifies the total area that will be available for all items in the group. For each item in the group, only the item height should be specified. The horizontal position and the width of each item will be taken from the `VGROUP` definition, and the vertical position will be calculated for you so as to provide equal space between each item in the group. `VGROUP` groups should be terminated with an `ENDVGROUP` command.

Example:

```
AUTOGROUP "Example group"
  VGROUP 10 12 100 70
    RADIOBUTTON 12    "Choice 1" %7 1
    RADIOBUTTON SAME  "Choice 2" %7 2
    RADIOBUTTON SAME  "Choice 3" %7 3
    RADIOBUTTON SAME  "Choice 4" %7 4
    RADIOBUTTON SAME  "Choice 5" %7 5
    RADIOBUTTON SAME  "Choice 6" %7 6
  ENDVGROUP
ENDGROUP
```

Available options:

None.

External interface

Overview of extension DLL interface

Glink allows your scripts to call functions in external DLL libraries, including all the standard Windows routines. To do this, you must write one or more additional DLL libraries to provide an interface between the script language and the actual functions you wish to call. These additional DLL libraries provide both a definition of the syntax of the extensions to the script language they implement and conversion between the data formats used by the script language and those used by the external routines.

Up to ten of these DLL libraries may be used simultaneously. Five of these are named conventionally from `GLSCREX0.DLL` to `GLSCREX4.DLL` (and are loaded automatically if present), while the other five may be referred to specifically by name in the script itself.

Using external functions in a script

From the script programmer's point of view, the extensions to the script language that are provided by the add-on DLL seem to be part of the script language itself. The only difference between verbs that are part of the native script language and those that are in the DLL is that the external verbs are marked with angle brackets (<>). For example, a script might wish to test whether or not another application is active (to check whether or not it needs to be started). A Windows developer would use the function `FindWindow()` directly, writing for example (in C):

```
if (!FindWindow("XLMAIN",NULL)) {
     WinExec ("EXCEL", SW_SHOWNORMAL);
}
```

Note that XLMAIN is the class name for the Excel main window in the above example. In that the script language does not provide a direct equivalent to the FindWindow() function we would define an equivalent function in an external DLL, calling it perhaps MyFindWindow. This function would be defined with three arguments, the first two being the same two arguments as are needed for the call itself, and the third used for saving the return value from the call so that the script can test the result. Later we'll look at how this is actually implemented, but for now we'll look at what would be written in the script itself:

```
<MyFindWindow> "XLMAIN" "" %1
if (%1 eqn 0) DOS "EXCEL"
```

These statements provide exactly the same functionality as the C code above, and assume that the MyFindWindow routine is in one of the standard (GLSCREXn) libraries. If the MyFindWindow routine had been in a specifically named library, for example WINAPI.DLL, then you could call it using the following syntax:

```
<WINAPI.MyFindWindow> "XLMAIN" "" %1
if (%1 eqn 0) DOS "EXCEL"
```

Note that you specify the name of the extension DLL as a simple file name with no extension when referencing a specific library in this way.

Programming external script functions

Leaving aside for the moment the question of how the syntax of the MyFindWindow verb is defined by the DLL, we can look at the actual implementation. In C this would look like this:

```
void WINAPI
    MyFindWindow (lpClass, lpWindow, lphWnd)
     LPSTR     lpClass;
     LPSTR     lpWindow;
     HWND FAR  *lpHwnd;
{
     if (!*lpClass)
          lpClass = NULL;
     if (!*lpWindow)
          lpWindow = NULL;
     *lphWnd = FindWindow(lpClass, lpWindow);
}
```

The same routine in Pascal would be written:

```
procedure MyFindWindow (lpClass, lpWindow : PChar;
                        var Result : Hwnd); export;
{$IFDEF WIN32} stdcall; {$ENDIF}
   begin
      if lpClass^  = #0 then lpClass  := nil;
      if lpWindow^ = #0 then lpWindow := nil;
      Result := FindWindow (lpClass, lpWindow);
   end {MyFindWindow};
```

Several points are worth noting at this stage. Firstly, none of the external routines may return a value (in Pascal terms, they are all procedures rather than functions). Any results that must be returned are returned in one or more of the arguments to the routine itself. The routine must also be defined as `WINAPI`, as is normal for routines in DLL libraries. Also, all arguments are passed by reference rather than by value - they will never be passed as integers for example, but instead passed as a pointer to an integer (or as a 'var' parameter if you are programming in Pascal).

Another small point to notice is that the Windows routine we are calling uses a null argument to specify that one of the parameters should not be used. In that the interface doesn't allow us to do this directly we've used a convention that interprets a null string as a signal to ignore the parameter, and the DLL routine makes the appropriate adjustment before calling the Windows routine.

The actual format and number of the parameters being passed to our new `<MyFindWindow>` script verb must of course be made known to the script compiler. Not only that, but it must be told that `<MyFindWindow>` is a valid verb. To do this, every external script extension library must have an extra routine called `GlinkVerb`. Every time an external verb is encountered in a script, the compiler will check with the `GlinkVerb` entry point in each external DLL that's found to see first of all whether the verb is recognized. If it is, then it will also check which parameters to expect.

Finally, you must remember to export the `MyFindWindow`, `GlinkVerb` and `GlinkValue`, if used, in the DEF file for a C module or in the `exports` statement for a Pascal module. Make sure they are case sensitive for `WIN32` DLL’s.

External interface

The syntax for the entry point looks like this in C:

```
int WINAPI
   GlinkVerb(VerbToParse, InputParams, OutputParams)
   LPSTR VerbToParse;  /* verb from script, from Glink    */
   LPSTR InputParams;  /* input parameter specification   */
   LPSTR OutputParams; /* output parameter specifications */
```

And like this in Pascal:

```
function GlinkVerb (VerbToParse,
                    InputParams,
                    OutputParams : PChar)
                    : integer; export;
{$IFDEF WIN32} stdcall; {$ENDIF}
```

If you are writing in C, then you must of course remember to export the routine in the DEF file for the DLL. In the same way, if you are writing in Pascal then you must remember to include the routine in the `exports` clause at the end of your source code. All the strings involved in the above are C-type null-terminated strings.

When the `GlinkVerb` routine is called the `VerbToParse` parameter will be filled out with the name of the verb in the script being compiled, less the **<>** angle-bracket delimiters. The name will also have been converted to uppercase. So in the example we have been using, the `GlinkVerb` routine will see `MYFINDWINDOW` as its input argument.

If the `GlinkVerb` routine does not recognize the verb being parsed, it should return a value of -1. If on the other hand it ***does*** recognize the verb, it should return the ordinal entry point of the routine that implements the verb in the DLL. The name will not be used to link to the routine at runtime, only the ordinal entry point number. At the same time, it must fill out the `InputParams` and `OutputParams` arguments, to define which parameters are to be used with the routine. These are specified with a single character for each parameter and are chosen from the following:

C	Null terminated string
H	Handle
I	Integer
L	Long integer
O	Script OK status (output only)
S	Structure

As we have already seen, the routine that implements the function in question will receive far pointers to each of the variable types specified rather than the actual values. So a 'C' type character parameter will be seen as a `LPSTR` / `PChar` type variable. Let's take a look at what this means in terms of the `MyFindWindow` example, which uses two character strings as input parameters and a window handle as its output parameter. These examples assume that the `MyFindWindow` routine has an entry point ordinal number of 101.

```
int WINAPI
    GlinkVerb(VerbToParse, InputParams, OutputParams)
     LPSTR VerbToParse;
     LPSTR InputParams;
     LPSTR OutputParams;
{
     if (!lstrcmp(VerbToParse, "MYFINDWINDOW")) {
          lstrcpy(InputParams,  "CC");
          lstrcpy(OutputParams, "H");
          return(101);
     }
     return (-1);
}
```

In Pascal:

```
function GlinkVerb  (VerbToParse,
                     InputParams,
                     OutputParams : PChar)
                     : integer; export;
{$IFDEF WIN32} stdcall; {$ENDIF}
   begin
      if strcomp(VerbToParse,'MYFINDWINDOW') = 0 then
        begin
           strcopy (InputParams,  'CC');
           strcopy (OutputParams, 'H');
           GlinkVerb := 101;
        end
      else
          GlinkVerb := -1;
   end {GlinkVerb};
```

It's probably a good idea at this point to follow exactly what happens when the `<MyFindWindow>` verb is now used in a script. At compile time the script compiler discovers that an external verb has been used and attempts to call the `GlinkVerb()` routine in any of the `GLSCREXn.DLL` libraries it finds (or in the specific named library if that syntax has been used). When it executes the `GlinkVerb()` routine we described above it will see that three arguments are required from what's returned by `GlinkVerb`, and also will see that the routine is entry point 101 in the DLL. It will check that three arguments have been provided, and remember these (the first two, being input arguments, may be coded in the script as constants; the last one being an output argument must be provided as a variable identifier).

At runtime the script executive will link to entry point 101 in the appropriate DLL and provide far pointers to the two strings specified. It will also provide a far pointer to a handle so that the DLL routine has somewhere to place the result of the call. After the routine has been called the script executive will take the value of the handle returned and convert it into a format suitable for further processing in the script.

Data types for the DLL

Here we will provide some specific information about the handling of the various data types used in external script libraries.

In general, all parameters provided for input arguments point to temporary holding areas - the data in these areas should not be modified (and such modification will not affect the actual variables used to supply the input in the script procedure).

Parameters provided for output arguments are also far pointers to temporary holding areas (rather than the actual output variables themselves). The script executive moves the data into the relevant output variables after the verb has been executed, supplying any necessary conversion at the same time. The following data types are available:

C: character data

Character data is passed to the external library as a far pointer to a null-terminated character array. Returned character data is copied back to the output variable, to a maximum of 255 characters (strings longer than this will be truncated).

H: handle data

For input parameters, the supplied input value will first be converted to a binary value (causing a runtime error if this is not possible). The external library will be passed a far pointer to this value. On output, the external verb will also be supplied with a far pointer to a binary value, and the script executive will convert this value back into string format before copying it to the output variable declared in the script procedure.

I: integer data

For input parameters, the supplied input value will first be converted to a 16-bit integer when running the 16-bit version of Glink otherwise to 32-bits (causing a runtime error if this is not possible). The external library will be passed a far pointer to this value. On output, the external verb will also be supplied with a far pointer to an integer, and the script executive will convert this integer back into string format before copying it to the output variable declared in the script procedure.

L: long integer data

For input parameters, the supplied input value will first be converted to a 32-bit binary value (causing a runtime error if this is not possible). The external library will be passed a far pointer to this value. On output, the external verb will also be supplied with a far pointer to a 32-bit binary value, and the script executive will convert this value back into string format before copying it to the output variable declared in the script procedure.

O: script OK status

This data type may be specified as an output parameter only. The value placed in this 16-bit variable by the external DLL should be zero or non-zero (false and true respectively), and will be reflected in the value of the script OK variable after the verb in question has executed. Note that when an 'O' parameter has been specified then although the external script procedure will be provided with a pointer to the return area, the call in the script itself will NOT contain a parameter at the equivalent position.

S: structure data

The input data in this case will normally be in a script variable, and the entire contents of that variable will be treated as data with unspecified contents. The external routine may access any of this data up to a limit of 255 bytes. When a structure variable is supplied as an output argument the external routine may place data anywhere inside the 255-byte holding area, and the entire 255-byte area will be copied to the target script variable, irrespective of contents (contrast with character variables, where the contents will only be copied up to and including the terminating null character). It is the responsibility of the external DLL not to move data outside the 255 bytes that are available for an output structure.

External values

Especially when you deal with the standard Windows routines, there are many symbolic constants that you might wish to code as names rather than directly as numbers. For example, a script using the `SW_MINIMIZE` constant is more understandable if you could code using that name rather than use '6' as a constant. Although it's possible to use the external verb functionality to collect such names this is still fairly complicated, and incurs an unnecessary overhead in that the value must be dynamically collected at run time). The external DLL facility caters for this need with an additional entry point, `GlinkValue`. This has the following syntax:

```
int WINAPI
    GlinkValue(ValueToParse, Result)
     LPSTR ValueToParse;  /* symbolic value, from Glink */
     LPSTR Result;        /* actual value */
```

In Pascal:

```
function GlinkValue (ValueToParse, Result : PChar)
                     : integer; export;
{$IFDEF WIN32} stdcall; {$ENDIF}
```

Here again, remember that the routine must be specifically exported in the DEF file or in the `exports` statement.

This entry point is supplied with the name collected from the script in the same way as the `GlinkVerb` entry point. However, all it has to do is to fill out the `Result` field with the relevant constant and return a non-zero value as the result of the function. If the supplied name is not recognized then `GlinkValue` should return a value of zero as its result.

The result supplied by `GlinkValue` should always be in null-terminated string format - numeric values should therefore be converted, using `wsprintf()` for example.

External values defined this way can be used anywhere in a script where a constant value would be accepted. If the syntax of the script requires a numeric value then the returned string will be checked at compile time for numeric content.

The `GlinkValue()` routine that would be needed to implement `SW_MINIMIZE` as an external constant (and thus let you use `<sw_minimize>` in a script) could look like this:

```
int WINAPI
    GlinkValue(ValueToParse, Result)
     LPSTR ValueToParse;
     LPSTR Result;
{
     if (!lstrcmp(ValueToParse, "SW_MIMIMIZE")) {
          wsprintf(Result, "%d", SW_MIMIMIZE);
          return(TRUE);
     }
     return(FALSE);
}
```

The same in Pascal:

```
function GlinkValue (ValueToParse, Result : PChar)
                     : integer; export;
{$IFDEF WIN32} stdcall; {$ENDIF}
  var
     S : string[12];
  begin
    if strcomp (ValueToParse, 'SW_MINIMIZE') = 0 then
```

```
        begin
           str (sw_Minimize, S);
           StrPCopy (Result, S);
           GlinkValue := 1;
           exit;
        end;
     GlinkValue := 0;
   end {GlinkValue};
```

Search rules

Extension script libraries should be located in the Glink user directory (specified with /U on the command line) or in the Glink directory (the same directory as GL.EXE). Only libraries containing both a `GlinkVerb()` and a `GlinkValue()` exported routine will be considered to be valid libraries.

Note that these exported routine names are case sensitive for WIN32 DLL's.

Glink will attempt to parse verbs and constants that don't supply a specific DLL name starting at the `GLSCREX0` library and continuing through to `GLSCREX4`. The first library to accept the verb or constant will be used; in other words a routine that exists in more than one library will be executed from the library with the lowest number.

Examples of extension DLLs

An extensive example of an external library supplying access to a number of Windows SDK routines is supplied in source with the Glink software (both as C and as Pascal). This can be used as a base upon which to build your own external libraries. They can be found in the C:\GLWIN\EXT directory and are called GLSCREX0.C and GLSCREX0.PAS.

Script examples

To make the use of some of the commands clearer, here are some examples of scripts that you could use for various purposes.

Simple login to bulletin board

There are several different varieties of these, but the script will in most cases look more or less the same, as all that is required is a simple dialog:

```
converse "FIRST name? " "john smith"
converse "(dots will echo): " $PASSWORD
quit
```

All we are doing here is to respond to the question about our name, wait for the question about our password (which finishes with the text "(dots will echo): " and then send our password. You will have to insert the appropriate strings for the prompts sent by your system, of course. Note that we are assuming here that this is a script that has been 'attached' to an entry in the dial directory, and is thus going to be started automatically as soon as we connect. The `$PASSWORD` here just means to send the password that we saved in our dial directory entry for this system. This also means that we can use this same script for all the systems that have prompts in exactly this form. We could also have used the same principle for the login name, using the `$LOGIN` variable.

More complex login

Let's take that script and make it a little more 'advanced'. We'll assume that the system we are talking to stops now and again and sends the --more-- prompt when it reckons our screen is full. (It isn't, of course; you can always look at what went past by going into the scrollback with the CTRL+PGUP key.) So we want to send a CR character every time we see that text. Also we know that the first command we want to send when we have been through the login procedure is "R". Let's see what that gives us:

```
pattern !1 "--more--"
when !1 sndline ""
converse "FIRST name? " "john smith"
converse "(dots will echo): " $PASSWORD
converse "Main Command" "r"
quit
```

Not much more complicated, but it's giving us a little more than the previous one, getting us completely into the system.

Login with error checking

Really the only thing missing with that script now is some kind of error checking, so as to make sure that the script stops properly if things go wrong en route. We'll finish up with something like:

```
 pattern !1 "--more--"
 when !1 sndline ""
 on timeout 15 goto Failed
 converse "FIRST name? " "john smith"
 converse "(dots will echo): " $PASSWORD
 on timeout 60 goto OKanyway
 converse "Main Command" "r"
:Okanyway
 quit
:Failed
 message "Script failed!"
 beep
 quit
```

You can see here the normal way of adding an error check, with the `ON` command. If for some reason we don't get the requisite prompt during the initial login phase, we'll just give up. If we login correctly but don't find the main command prompt, we'll just carry on anyway, though, in that we are already logged in.

'Event-driven' login

A somewhat more 'sophisticated' way of preparing your login script is by letting events decide what to send. This technique is especially useful when talking to systems that don't always ask the same questions in the same order, or are otherwise 'difficult' to talk to for one reason or another. Let's look at a login script for getting to BIX via the Norwegian Datapak service:

```
pattern  !1 "NUI?"
when  !1 sndline "N00789PASSWD"
pattern  !2 "ADD?"
when  !2 sndline "A031069"
pattern  !3 "please log in:"
when  !3 sndline "bix"
pattern  !4 ".More.."
when  !4 sndline ""
pattern  !5 "Name?"
when  !5 sndline "jsmith"
pattern  !6 "Password:"
when  !6 sndline $PASSWORD
pace  2
strip yes
delay 2
sndline  ""
on timeout 90 goto Failed
receive  "^J:"
quit
:Failed
disconnect
message "Sorry, no luck!"
quit
```

This one is slightly more complex, but is worth having a look at. What it's doing essentially is to set up a number of things that it knows it will see on the way into the system, and predefining how it will react to each one. All this before doing anything else at all. Once these have been set up we can start work. Here we set up a number of parameters that are particular to the service we are logging into, and then just send a CR (`sndline ""`) to kick things off. Each time the appropriate message arrives we send the appropriate answer, and all the main body of the script has to do is to wait for that final prompt that signals success.

Menu-controlled script

This particular example is taken from the Bull environment, and provides menu-controlled choices of logins to different applications and functions. Hopefully it can give you some ideas of how you can use Glink to provide a friendly popup-style interface to some of your most-used applications. The script is designed to run on a Glink terminal connected to *Server6*, where the script would be activated using the following *Server6* menu command:

```
>xx a PCS this.scr;V78 DSS
```

where `'this.scr'` is the name of the PC file containing the following script:

```
rece "VIP7804"
rece "[W"
assi %1 ""
:MENU
   time 30
   menu   "<< GCOS8 menu >>"
   mop    "     Start GREDS           "  goto GREDS
   mop    "     Start PCF             "  goto PCF
   mop    "     Collect a file        "  goto GETONE
   mop    "     Send a file           "  goto SENDONE
   mop    "     Manual login          "  goto WAIT
   mtext  " (Recall menu with Alt+O)  "
   mop    " <esc> back to Server6     "  goto ALLDONE
   domenu
:ALLDONE
   if (%1 eq "") goto STOPV78
   sndl "$*$DIS"
   rece "[W"
   dten 5
:STOPV78
   sndl "^[^^"
   rece "[W"
   dten 5
   sndl "^[T"
   quit
:GREDS
   gosub LOGIN
   sndl "GREDS"
   goto WAIT
:PCF
   gosub LOGIN
   sndl "PCF"
   goto WAIT
```

```
:GETONE
   input %9 "File to collect: "
   gosub LOGIN
   sndl "FTRA PC7800"
   getfile FTRA %9
   goto WAIT
:SENDONE
   input %9 "File to send: "
:SEND0
   EXISTS FILE %9
   if OK goto SEND1
   beep
   input %9 "File not found, try again: "
   goto SEND0
:SEND1
   gosub LOGIN
   sndl "FTRA PC7800"
   putfile FTRA %9
   goto WAIT
:LOGIN
   if (%1 eq "DPS8") return
   sndl "$*$CN TSS,DPS8"
   rece "USER ID --"
   menu "TSS password"
   invi %9 "Current password please: " 12
* hide the password
   show "^[sh"
   sndl ("GAR$" %9)
   rece "*"
   return
:WAIT
   assi %1 "DPS8"
   online
   goto MENU
```

The script above when activated waits for G&R/V78 to be started as the second command in the menu line, and then it presents a menu for various login and functional dialogs. After completing each menu function, the script waits for the script to be reactivated by the user with the `ALT+O` key. At that point, it returns to the menu.

Running VBScript or JScript files

If the Microsoft Script Control is installed on your PC then Glink automatically runs VBScript or JScript files if the file name is of the form `*.VBS*` or `*.JS*`.

The VBScript or JScript file can be started in the same ways you can start a Glink script, e.g. assigned to macros using ^*myscript.js and then used on the buttonbar or user menu.

```
SETMACRO 22 "^*MyScript.JS"
BUTTON 2 MACRO-22 "Special"
```

Glink will call a function called "`Main`", which must be present in the script file, and then wait for the ScriptControl engine to terminate execution before continuing.

```
//***************
// JScript Demo *
//***************
function Main()
{
// do something here
}

'*****************
' VB Script Demo *
'*****************
Sub Main()
' do something here
End Sub
```

Inheriting the GlinkApi object

The Glink.GlinkApi and Glink.Auto objects of the current session are passed on to the script and are available under the names of "`GlinkApi`" and "`Auto`", e.g:

(VBS script)

```
'*****************
' VB Script Demo *
'*****************
Const TITLE = "Glink VBScript Demo"

Sub Main()
  Dim startPt
  Dim endPt
  Dim s

  set startPt = GlinkApi.getCursor
  startPt.x = 1
  set endPt = GlinkApi.getCursor
  endPt.x = endPt.x  - 1

' pick text from start of line to cursor pos (-1)
  S = GlinkApi.getString (startPt, endPt)

  Call Msgbox("Current line of text is: " & S,
vbInformation, TITLE)

End Sub
```

(VScript script)

```
//***************
// JScript Demo *
//***************
function Main()
{
  var startPt;
  var endPt;
  var s;

  startPt = GlinkApi.getCursor();
  startPt.x = 1;
  endPt = GlinkApi.getCursor();
  endPt.x = endPt.x  - 1;

// pick text from start of line to cursor pos (-1)
  S = GlinkApi.getString (startPt, endPt);

  endPt = GlinkApi.getCursor();
  GlinkApi.sendKeys ("1", endPt);
  GlinkApi.sendCommandKey (55); // TRANSMIT key
}
```

Please note that it is **not recommended** that you call the `GlinkApi.scriptFile()` or `GlinkApi.scriptCommand()` methods from a VBS or Jscript as you will most certainly encounter synchronization and even reentrancy problems because of their simultaneous execution.

Passing input parameters

You pass input parameters to the VBScript or JScript in exactly the same way as you would a Glink script using parenthesis or using the `PARAM` script command:

```
"Myscript" param1 param2
```

The `Main()` procedure will receive these parameters as one string. You then need to add this input parameter string to the `Main()` procedure:

```
Function Main(Param)

  Msgbox Param, vbInformation, "Title"
  Main = "Thank you for calling"
```

```
End Function
```

Here is an extract taken from the `vbs.sub` and `word.vbs` files delivered in the `$GLINK\Scripts\Demo directory`:

(Glink Script, vbs.sub)

```
Assign %demodir "c:\glwin\scripts\demo"
Call ("'" %demodir "\word.vbs" "' " %demodir
"\glinkpro.doc")
```

or

```
Assign %demodir "c:\glwin\scripts\demo"
Param (%demodir "\glinkpro.doc")
Call (%demodir "\word.vbs")
```

(VBS script, word.vbs)

```
Sub Main(DocName)
  Dim myWord

  GlinkApi.setVisible (false)

  Set myWord = CreateObject("Word.Application")
  myWord.Visible = True
  myWOrd.Documents.Open DocName

  ' do something here....
  myWord.Quit
  set myWord = nothing

End Sub
```

Return values

Return values from a VBScript or JScript are returned to Glink in the $GPARAM internal script variable. You need to define `Main()` as a function and return a value. The VBS example shown here will display `"Hello world"` in a message box and when control is returned to Glink `"Thank you for calling"` will be displayed.

(Glink script)

```
Param ("Hello world")
Call ("param.vbs")
Show $GPARAM
```

(VBS script)

```
Function Main(Param)

  Msgbox Param, vbInformation, "Title"
  Main = "Thank you for calling"

End Function
```

Configuration file format

This table lists the data elements in the Glink configuration file, and may be used for 'expert' manipulation of the file contents from scripts, using the CRDB, CREAD, CFIX and CFXW script commands. The following data types are used:

byte	single 8-bit byte
char	single 8-bit character
bool	single byte, 0 = false, 1 = true
int	2 bytes, low order byte first
char[n]	array of n characters
int[n]	array of n integers
string[n]	Pascal string format, leading byte contains string length

Some examples of how to manipulate the various types of data are provided at the end of this appendix.

Units throughout are as in GLINK setup menus unless otherwise noted.

> **NOTE**
> If you are running on a system with long file names and Glink is enabled for use of these, then file names in the configuration file are saved on an additional file with the same name as the original configuration file and a suffix of `'.ini.glinkdata'` (for example `C:\GLWIN\DEF.glinkconfig.ini.glinkdata`). This type of field is marked with a note in the form ***(xx*)*** in this list, where ***xx*** is the identifier used for the option in the `INI` file. These fields may not be modified with `CFIX` or `CFXW`.

Offset	Length	Format	Contents
0	2	int	Baud rate divided by 10
2	1	byte	Parity/format 0=7E. 1=8N, 2=7O, 3=8E, 4=8O
3	1	bool	Initial echoplex setting
4	1	byte	Current emulation mode 0=VIP7800, 1=ANSI, 2=Prestel, 3=Minitel, 4=VT102/220, 5=VIP7700, 6=DKU7107, 7=IBM3270, 8=DKU7102, 9=IBM5250, 10=IBM3151
5	1	byte	7200 attributes 0=no, 1=yes, 2=extended
6	1	bool	Initial roll mode setting
7	4	char[4]	Enquiry answer string
11	1	bool	TAPI controls modem
12	1	bool	Initial auto lf setting
13	2	int	Comms pacing
15	1	bool	Hold DTR in local mode
16	1	byte	Flow control 0=none, 1=DTR, 2=RTS, 3=Xon-Xoff
17	1	bool	'Double graphics'
18	1	byte	Cursor type 0=line, 1=blink, 2=block
19	1	byte	Screen update type 0=direct, 1=retrace, 2=Bios
20	1	byte	Default screen attribute
21	1	byte	Default status line attribute
22	14	-	<not used>
36	1	byte	Alarms after file transfer
37	1	byte	Wait after transfer 0=never, 1=always, 2=when failed
38	1	bool	Silent mode

Configuration file format

Offset	Length	Format	Contents
39	2	int	Commsport 0 = COM0, 1-4 = COM3-4, 5-12 = SERIAL1-8, 13-14 = MM4 COM3-4, 15-16 = EVEREX COM3-4
41	1	bool	Reset on Clear
42	1	char	Kermit host quote character
43	1	bool	Short screen dump
44	1	bool	Suppress high intensity
45	1	byte	Kermit pacing
46	1	byte	Underline simulate attribute
47	1	byte	Kermit timeout
48	1	byte	Kermit retries
49	1	bool	Inverted screen option
50	1	bool	Suppress clock update
51	1	bool	Escape is F7
52	4	string[3]	8bit keyboard file
56	1	-	<not used>
57	1	-	<not used>
58	1	bool	Suppress error messages
59	1	bool	Arabic mode
60	1	bool	EGA underline option
61	1	bool	Allow long packets
62	1	bool	Suppress welcome message
63	1	bool	File overwrite option
64	1	bool	ASC expand blank lines
65	1	char	ASC upload pace character
66	2	int	ASC upload character pacing
68	2	int	ASC upload line pacing

Offset	Length	Format	Contents
70	1	byte	ASC upload CR translation 0 = strip, 1=CR, 2=LF, 3=CRLF
71	1	byte	ASC upload LF translation 0 = strip, 1=CR, 2=LF, 3=CRLF
72	1	byte	0=EOT delimiter, 1=ETX delimiter
73	1	bool	Add CRLF in SSM option
74	1	bool	Initial space suppression option
75	1	bool	Initial TX-RET setting
76	1	bool	0=CHAR initially, 1=TEXT initially
77	1	bool	Auto tabbing option
78	1	bool	Extended status option
79	1	bool	Initial block mode setting
80	1	byte	Pre-print controls 0=CR, 1=CRLF, 2=CRFF, 3=CRVT
81	1	byte	Post-print controls 0=CR, 1=CRLF, 2=CRFF, 3=CRVT
82	1	-	<not used>
83	1	bool	Host Xoff option
84	1	bool	Suppress parity errors
85	4	string[3]	Seven-bit keyboard file
89	1	bool	Eight-bit host option
90	41	string[40]	Modem init string
131	17	string[16]	Modem dial string
148	17	string[16]	Modem on-hook string
165	17	string[16]	Modem off-hook string

Configuration file format

Offset	Length	Format	Contents
182	1	byte	Communications interface 9=G&R NetBIOS, 13=IBM LANACS, 16=Eicon, 19=raw NetBIOS, 31=Windows, 33=DNTD gateway, 38=Windows Sockets, 40=none, 46=DNTD/SPX, 47=IC, 48= G&R SPX, 49=Atlantis TSA V8, 51=ICC, 52=Shiva V8, 53=Cirel VTI3, 54=Cirel FPX, 55=Eicon TGX, 57=Telephony, 58=GLAPI
183	1	bool	Constant speed modem
184	1	byte	<not used>
185	1	bool	Start in dialing directory
186	2	int	Commport base address
188	1	byte	Commport IRQ number
189	1	byte	Wait before dialing
190	1	byte	Pause between dials
191	4	string[3]	Keyboard layout file
195	2	int	Xon-Xoff timeout
197	1	bool	Add Ctrl-Z option
198	1	bool	ANSI use high intensity
199	2	int	'noise level'
201	2	int	Printer number -1=print on file, 0=LPT1, 1=LPT2, 2=LPT3
203	1	bool	Report printer busy
204	2	int	Comms buffer size
206	31	string[30]	DNTD Host InsID
237	3	-	<not used>
240	1	bool	BIOS scroll option
241	650	<internal>	macro strings
891	25	-	<not used>

Offset	Length	Format	Contents
916	1	byte	Default download protocol 0=ascii, 1=Kermit text, 2=Kermit binary, 3=Xmodem, 4=Ymodem, 5=Ymodem batch, 6=Ymodem-G, 7=Modem7, 8=Telink, 9=Zmodem, 10=CIS-B
917	1	bool	Strip parity
918	1	bool	CLR saves in scrollback
919	1	bool	Host name in status line
920	5	char[5]	colors first level window
925	5	char[5]	colors second level window
930	5	char[5]	colors third level window
935	5	char[5]	colors fourth level window
940	1	bool	no pseudocolumn 81
941	62	byte[62]	VIP attribute mapping table
1003	1	bool	Auto LF in
1004	2	-	<not used>
1006	1	bool	Simple dialing option
1007	1	bool	CTS handshake
1008	1	-	<not used>
1009	38	int[19]	User config menu choices
1047	1	bool	Clock shows time online
1048	1	-	<not used>
1049	1	bool	<not used>
1050	1	bool	Call logging
1051	1	bool	Dialing directory password disable
1052	1	bool	Comments on call log
1053	1	bool	Lock dialing directory
1054	1	byte	Sync poll address

Configuration file format

Offset	Length	Format	Contents
1055	1	byte	Mouse sensitivity X
1056	1	byte	Mouse sensitivity Y
1057	2	int	Script buffer size
1059	2	int	Max script variables
1061	25	string[24]	Print on file name (PN*)
1086	25	string[24]	Pre-print filename (PN1*)
1111	25	string[24]	Post-print filename (PN2*)
1136	25	-	<not used>
1161	1	bool	Constant reminders
1163	2	int	Max Kermit packet size
1164	2	int	Max Kermit window size
1166	1	bool	ANSI 25-line mode
1167	1	bool	Optimize comms option
1168	1	bool	Enter always transmits
1169	1	char	Kermit packet header character
1170	1	bool	Non standard FTRAN
1171	1	bool	Remove printer deletes
1172	1	byte	X.25 user group
1173	1	char	Telnet break character
1174	1	bool	Ctrl-Z is end of file
1175	2	int	Max menu items in scripts
1177	1	byte	Printer translation option 0=US, 1=UK, 2=HOL, 3=FIN, 4=FR 5=FRC, 6=GER, 7=IT, 8=NOR, 9=SPA, 10=SWE, 11=SWI, 12=DEN, 13=JAP
1178	1	bool	Reset modem on dial
1179	1	bool	Save aborted downloads
1180	25	string[24]	X.25 user data

Offset	Length	Format	Contents
1205	25	string[24]	Server target (LANACS)
1222	1	byte	Logical channel
1223	1	bool	Capture delimiter is CR
1224	1	bool	Suppress printer transliteration
1225	1	byte	Eicon (and other) port number
1226	61	string[60]	Appointments directory (AD*)
1287	1	byte	UVTI interrupt
1288	1	bool	SISO encoding for eightbit
1289	1	bool	Suppress status line
1290	1	byte	Menu shadow type
1291	32	byte[32]	Configuration locks
1323	1	bool	No file name translation
1324	1	bool	Ignore remote commands
1325	1	byte	OSI interrupt number
1326	1	bool	Non-linear forms
1327	1	bool	Small transfer window
1328	33	string[32]	X.25 facilities
1329	1	bool	Enable FIFO on UART
1362	1	bool	Disallow status line lock
1363	1	byte	Mouse cursor type 0=Default, 1=Diamond, 2=Block, 3=Blink
1364	1	bool	Destructive backspace
1365	25	string[24]	Dial abbreviation A
1390	25	string[24]	Dial abbreviation B
1415	25	string[24]	Dial abbreviation C
1440	25	string[24]	Dial abbreviation D
1465	25	string[24]	Dial abbreviation E

Offset	Length	Format	Contents
1490	25	string[24]	Dial abbreviation F
1515	1	bool	Use DPS8 compression (FTRAN)
1516	2	int	X.25 Permanent VC number
1518	17	string[16]	X.25 calling address
1535	33	string[32]	LAN server name
1568	1	bool	Typeahead mode option
1569	1	bool	Print log all line wrap
1570	1	-	<not used>
1571	41	string[40]	TSM directory (TSM*)
1612	1	bool	TSM enable
1613	1	byte	Kermit tab expansion (0=off)
1614	17	string[16]	Extra modem connect string
1631	17	string[16]	Extra modem OK string
1648	1	bool	Override host packet size
1649	1	byte	Cirel/Atlantis card number
1650	1	byte	Cirel cluster number
1651	41	string[40]	Extra modem init string
1692	1	bool	Two-wire sync connection
1693	1	bool	Long quiescent frame
1694	1	byte	Atlantis interrupt number
1695	1	bool	Ignore carrier status
1696	1	byte	Physical channel (Atlantis)
1697	33	string[32]	ANSI answerback
1730	15	string[14]	Telnet reply to enquiry
1745	1	byte	Windows start mode 0=default, 1=normal, 2=icon, 3=maxi
1746	1	byte	Windows start X position

Offset	Length	Format	Contents
			0=default, 1=left, 2=right, 3=center, 4=current
1747	1	byte	Windows start Y position 0=default, 1=top, 2=bottom, 3=center, 4=current
1748	1	byte	Windows initial font size, X
1749	1	byte	Windows initial font size, Y
1750	1	-	<reserved>
1751	1	bool	DEC Pathworks
1752	1	bool	Windows menu bar
1753	2	int	Windows printer timeout
1755	1	bool	Windows print direct
1756	1	-	<not used>
1757	17	-	<not used>
1774	1	bool	Windows print draft mode
1775	1	bool	Windows popup dial
1776	1	bool	Telnet binary session
1777	1	byte	Windows exit confirmation 0=always, 1=when connected, 2=never
1778	1	bool	Telnet use CR-NUL
1779	1	bool	Windows host name in title bar
1780	1	byte	Kermit quoting 0=default, 1=ON, 2=OFF
1781	4	string[3]	Kermit keyboard file
1785	1	bool	Windows ASCII OEM transfer
1786	1	-	<reserved>
1787	1	-	<reserved>
1788	25	string[24]	Host file transfer command
1813	1	bool	Windows VT102/220 function keys
1814	1	bool	ANSI erase with default background

Configuration file format

Offset	Length	Format	Contents
1815	1	bool	Use internal Zmodem
1816	1	bool	PNC mode
1817	1	bool	Use Zmodem compression
1818	1	bool	Use Zmodem recovery
1819	1	bool	Use Zmodem transparent
1820	1	bool	Use compressed context files
1821	1	byte	Disconnect action 0=None, 1=Terminate, 2=Reconnect
1822	1	byte	TCP/IP protocol 0=Telnet, 1=SNI, 2=NCR, 3=DNTD gateway, 4=Raw TCP/IP, 5=DIWS gateway, 6=DSA gateway, 11=rlogin, 12=TNVIP, 13=TN3270, 14=TN5250
1823	2	int	Screen update threshold (chars)
1825	1	bool	Use INT5A for Eicon NABIOS
1826	1	bool	Function keys send CR
1827	1	-	<reserved>
1828	1	byte	XON character
1829	1	byte	XOFF character
1830	50	<internal>	ANSI mappings
1880	1	-	<reserved>
1881	1	byte	Initial font 132 cols, X
1882	1	byte	Initial font 132 cols, Y
1883	1	byte	Initial font 40 cols, X
1884	1	byte	Initial font 40 cols, Y
1885	1	-	<reserved>
1886	64	<internal>	Macro offsets
1950	2	word	Typeahead wait
1952	2	word	Typeahead idle

Offset	Length	Format	Contents
1954	2	int	TX-EDT buffer size
1956	1	-	<reserved>
1957	1	byte	Printer character set 0=PC/OEM, 1=ANSI, 2=HP Roman, 3=PC/OEM (x)
1958	16	byte[16]	Configuration locks
1974	1	-	<reserved>
1975	1	byte	VIP7700 compatibility mode 0=normal, 1=ITT Courier, 2=Thomas box
1976	1	-	<not used>
1977	1	bool	Don't show spaces in 7700 forms
1978	1	bool	Restrict 7700 cursor movement
1979	1	bool	X.25 raw mode
1980	1	bool	Telnet use IP for break
1981	13	string[12]	Host alarm sound file (HW*)
1994	13	string[12]	Emulator alarm sound file (GW*)
2007	13	string[12]	File transfer sound file (XW*)
2020	13	string[12]	Connect sound file (CW*)
2033	13	string[12]	File transfer failed sound file (BW*)
2046	2	int	Saved window X position
2048	2	int	Saved window Y position
2050	2	int	Saved scrollback window X position
2052	2	int	Saved scrollback window Y position
2054	1	byte	Initial screen width for ANSI mode
2055	1	byte	Initial screen width for VT102/220 mode
2056	1	byte	Initial screen width for VIP7700 mode
2057	1	byte	Initial screen width for VIP7800 mode
2058	1	byte	Initial screen width for Minitel mode

Offset	Length	Format	Contents
2059	1	byte	Initial screen width for Prestel mode
2060	1	byte	Initial screen width for IBM3270 mode
2061	1	byte	Initial screen width for DKU7107 mode
2062	1	byte	Screen update threshold (lines)
2063	1	-	<not used>
2064	1	bool	Play sound files asynchronously
2065	1	bool	Host autotabbing
2066	1	byte	Initial number of buttons on bar
2067	1	byte	Initial button rows
2068	1	bool	Use status bar
2069	1	bool	Use toolbar
2070	1	bool	Sticky roll mode
2071	1	byte	TCP block size (divided by 16)
2072	1	byte	TCP block delay (milliseconds)
2073	1	bool	Number of items in toolbar
2074	40	int[20]	Toolbar bitmap references
2114	40	int[20]	Toolbar function references
2154	1	byte	Minimum length to save for EDIT mode
2155	1	-	<reserved>
2156	1	byte	IBM3270 model 0=3279-2, 1=3279-3, 2=3278-1, 3=3278-2, 4=3278-3, 5=3278-4, 6=3278-5, 7=3287-1, 8=3279-2E, 9=3279-3E, 10=3278-1E, 11=3278-2E, 12=3278-3E, 13=3278-4E, 14=3278-5E
2157	1	byte	IBM3270 transliteration 0=international, 1=UK, 2=US, 3=Swedish-Finnish, 4=French, 5=French-Canadian, 6=Austria-Germany, 7=Italian, 8=Denmark-Norway, 9=Spain, 10=Swedish-Finnish alternate, 11=Belgium, 12=Denmark-Norway alternate, 13=Japan, 14=Brazil, 15=Portugal, 16=Spain alternate,

Offset	Length	Format	Contents
			17=Spanish-speaking, 18=Austria-Germany alternate
2158	1	-	<not used>
2159	2	int	Saved window horizontal size
2161	2	int	Saved window vertical size
2163	2	int	Saved scrollback window horizontal size
2165	2	int	Saved scrollback window vertical size
2167	2	-	<not used>
2169	17	string[16]	IBM3270 LU name
2186	96	internal	Screen color mappings
2282	1	bool	Use special Winsock fix for PC-NFS
2283	1	bool	Preserve keyboard state
2284	1	bool	Respect OEM keyboard mappings
2285	1	bool	Don't check comm port existence
2286	1	bool	Use line-oriented screen marking
2287	1	bool	Use comms notifications
2288	1	bool	Use toolbar tips
2289	1	bool	Variable fields in 3D
2290	1	bool	DKU blink/blank with ^/~
2291	1	bool	DKU show ^/~ in blink/blank
2292	1	bool	DKU cursor straight up/down
2293	1	bool	DKU allow cursor out of field
2294	1	bool	DKU use extended character set
2295	1	bool	DKU allow lowercase to host
2296	1	bool	DKU new line after XMT
2297	1	bool	DKU SDP attributes
2298	1	bool	DKU wrap on page overflow

Offset	Length	Format	Contents
2299	1	byte	DKU color mode 0=1M, 1=4A, 2=4B, 3=7Q, 4=7G
2300	3	string[2]	DKU printer ID
2303	2	int	DKU printer columns
2305	2	int	DKU printer lines/page
2307	2	int	DKU printer cps
2309	12	byte[12]	DKU unshifted FKC types 0=Send, 1=Send Page, 2=Display
2321	24	char[12,2]	DKU unshifted FKC function codes
2345	1	byte	DKU model 0=DKU7107, 1=DKU7211
2346	1	bool	Font follows window
2347	1	bool	Use Windows fonts
2348	1	bool	Use caption bar
2349	1	byte	TGT type 0=X25, 1=VIPLS, 2=VIPX25, 3=FPM
2350	1	bool	Use print ctl for host data
2351	33	string[32]	Windows font name
2384	1	bool	New line after transmit
2385	1	bool	ANSI BBS compatibility mode
2386	1	bool	Don't use NetBIOS callbacks
2387	1	bool	DKU use SI/SO for printing
2388	1	byte	Screen paint algorithm 0=spread, 1=truncate, 2=squash
2389	2	int	Case 6000 device type
2391	1	bool	Suppress use of CTL3D
2392	1	-	<reserved>
2393	12	byte[12]	DKU shifted FKC types 0=Send, 1=Send Page, 2=Display

Offset	Length	Format	Contents
2405	24	char[12,2]	DKU shifted FKC function codes
2429	1	bool	Printer orientation 0=Default, 1=Auto, 2=Portrait, 3=Landscape
2430	1	byte	<reserved>
2431	1	byte	Stay-on-top option 0=Normal, 1=On top when icon, 2=Always on top
2432	1	byte	Windows font style Top bit = italic, bottom 7 = weight / 10
2433	1	byte	Host printing usage 0=GUI, 1=Windows(text), 2=File(text)
2434	33	string[32]	Printer font name
2467	1	byte	Printer font style Top bit = italic, bottom 7 = weight / 10
2468	1	byte	Printer select 0=session, 1=current, 2=permanent
2469	81	string[80]	Printer name,driver,port (PI*)
2550	2	int	Printer font height (tenths of a point)
2552	4	long	SPX network number
2556	1	bool	<reserved>
2557	1	bool	DKU wraparound tabbing
2558	1	bool	Reverse NumLock action for app keypad
2559	1	bool	DKU extensions for VIE
2560	2	int	Ggate keepalive interval (seconds)
2562	1	bool	DKU right justify ignores spaces
2563	67	string[66]	Download directory (DD*)
2630	67	string[66]	Upload directory (UD*)
2697	17	string[16]	Rlogin userid
2714	1	bool	Roll mode (DKU7107 only)
2715	1	byte	Initial screen width for DKU7102 mode

Offset	Length	Format	Contents
2716	24	byte[24]	Macro mappings for DKU function keys
2740	2	int	Print left margin
2742	2	int	Print right margin
2744	2	int	Print top margin
2746	2	int	Print bottom margin
2748	1	byte	Print margin units 0=dots, 1=inches, 2=cm, last two saved in hundredths
2749	33	string[32]	Socks server address
2782	1	bool	Suppress 3270 printing
2783	33	string[32]	Alternate server address for Ggate
2816	2	int	Connect timeout for Winsock TCP/IP
2818	1	bool	Use random connect for Ggate
2819	2	int	Delay before alternate connect for Ggate
2821	1	byte	Initial emulation mode (Windows only) 0=VIP7800, 1=ANSI, 2=Prestel, 3=Minitel, 4=VT102/220, 5=VIP7700, 6=DKU7107, 7=IBM3270, 8=DKU7102, 9=IBM5250, 10=IBM3151
2822	1	bool	Tab key sends HT in DKU7102 mode
2823	1	bool	Extended TN3270 enable
2824	1	bool	TN3270 LU name is associated
2825	1	char	Kermit control quote character
2826	1	-	<reserved>
2827	41	string[40]	Secondary TSM forms directory (TS2*)
2868	1	bool	Use context style help only
2869	1	byte	Turn delay in tenths
2870	1	bool	Use immediate clipboard rendering
2871	1	bool	Suppress print logging of FF char in 7800

Offset	Length	Format	Contents
			mode
2872	1	bool	Sets font for status bar (Windows only) (0=ANSI variable font, 1=system font, 2=device default font, 3=ANSI fixed font, 4= system fixed font, 5=OEM fixed font)
2873	1	bool	Suppress all use of EM as single shift
2874	1	bool	Apply 'Norwegian ASCII' to text as well as controls.
2875	1	bool	Send $*$ messages as data rather than command (Ggate)
2876	1	bool	Use keyboard lock where typeahead would normally be applied
2877	1	bool	Apply 'EmuLink' logic to printed output from host
2878	1	bool	Print fields even though they do not have a 'printable' attribute
2879	1	byte	DDE comms wait timeout (milliseconds)
2880	1	bool	Suppress AUX port printing
2881	1	bool	Capture line wraps
2882	1	bool	Smooth scroll in scrollback
2883	1	bool	Suppress 3270 locked keyboard
2884	1	byte	Use Print SS2 characters (0=none, 1=normal, 2=special)
2885	1	byte	IBM5250 model (0=IBM3179_2, 1=IBM3180_2, 2=IBM3196_A1, 3=IBM3477_FC, 4=IBM3477_FG, 5=IBM5251_11, 6=IBM5291_1, 7=IBM5292_2, 8=IBM5555_C01, 9=IBM5555_B01, 10=IBM3812_1, 11=IBM5553_B01)
2886	1	byte	IBM3151 model (0=IBM3151_11, 1=IBM3151_31, 2=IBM3151_41, 3=IBM3151_51, 4=IBM3151_61)
2887	36	byte[36]	IBM3151 FKC function codes
2923	36	byte[36]	IBM3151 FKC types (0=Send, 1=Display)

Configuration file format

Offset	Length	Format	Contents
2959	1	bool	Use colors for printing
2960	1	bool	<not used>
2961	1	bool	Local printing usage 0=GUI, 1=Windows(text), 2=File(text)
2962	1	byte	Print lines per page
2963	2	int	Print line spacing (0=auto, 65535=fill page, n=exactly)
2965	1	byte	Print line spacing unit (0=Dot, 1=Inch, 2=Cm, 3=Lpi, 4=Lpcm)
2966	1	byte	Print characters per line
2967	2	int	Print character spacing (0=auto, 65535=fill page, n=exactly)
2969	1	byte	Print character spacing unit (0=Dot, 1=Inch, 2=Cm, 3=Cpi, 4=Cpcm)
2970	2	Int	Scrollback pages
2972	96	Internal	Print color mappings
3068	1	Bool	Copy/paste with Ctrl+C/V
3069	1	Char	TCS ACK, positive ACK (default 'a')
3070	1	Char	TCS PGOF, logical NAK (default 'b')
3071	1	Char	TCS NAK, physical NAK (default 'c')
3072	1	Byte	TCS enable (0=inactive, 1=enable, 2=disable)
3073	41	string[40]	TCS directory (TCS*)
3114	1	<internal>	Config version update state
3115	25	String[24]	Print on file name (PD*) (physical)
3140	1	Bool	Print to file (physical)
3141	1	Bool	Print to file (Windows)
3142	1	Byte	IND$FILE command (0=current, 1-127=tab 128-255=backtab)
3143	1	Bool	Don’t strip underline in 3D variable fields

Offset	Length	Format	Contents
3144	1	Bool	Don’t use Windows 3D colors on variable fields
3145	1	Bool	Don't fix DKU attributes
3146	1	Byte	Enforce check for data in last line on 7800 IL command (0x0002 from host, 0x0001 from keyboard)
3147	1	Bool	Paste/Upload insert new line
3148	1	Byte	Paste/Upload max chars
3149	1	Bool	Paste/Upload wrap on last word
3150	1	Byte	<not used>
3151	1	Bool	Stops PPP auto-dialup box display `(Win32)`
3152	1	Bool	Local file overwrite
3153	1	Char	Default monetary character for numeric edited fields (7800)
3154	1	Byte	Number of recently used items to remember (default=10)
3155	1	Byte	3270 numeric checking (0=none, 1=strict, 2=emulator, 3=relaxed)
3156	1	Bool	Use XMT (not CR) for Enter in TX-RET mode when Enter=XMT
3157	1	Bool	Suppress hide attribute for 3270
3158	32	Internal	3D color mappings
3190	1	Bool	Don’t use Windows caret in variable fields
3191	1	Byte	Font options, 0x01 short underline, 0x02 Zero w/dot, 0x04 Zero w/slash
3192	1	Byte	Ruler type, 0=none, 1=horiz, 2=vert, 3=crosshair
3193	7	String[6]	Double-click delimiters (DC*)

Offset	Length	Format	Contents
3200	1	Bool	DKU extended ANSI SGR attributes
3201	1	Byte	Use printer timeout, 0x01 local, 0x02 host
3202	1	Bool	Suppress leading blank print page
3203	1	Bool	Don't allow cursor move with left mouse click
3204	1	Bool	Don't let host move 3270 graphics cursor
3205	1	Bool	Don't send 3270 graphic mouse click to host
3206	1	Byte	3270 graphics cursor type (0=bw target, 1=XOR target, 2=bw diagonal, 3=XOR diagonal, 4=bw cross, 5= XOR cross)
3207	8	byte[8]	Security options
3215	1	Bool	5250 Field exit keys on numeric pad
3216			etb/etx on 7800 block transmit (0=std, 1=etb, 2=etx)
3217	1	Bool	Use old toolbar format
3218	41	String[40]	Wallpaper file name (WP*)
3259	2	Int	Left window frame margin
3261	2	Int	Right window frame margin
3263	2	Int	Top window frame margin
3265	2	Int	Bottom window frame margin
3267	1	Bool	Margin is specified as percent
3268	1	Bool	Stretch wallpaper to fit screen
3269	1	Bool	Don't scroll wallpaper with text
3270	41	String[40]	Frame wallpaper file name (FP*)
3311	1	Bool	Keep aspect ratio (wallpaper)
3312	1	Bool	Keep aspect ratio (frame wallpaper)
3313	1	Bool	Center wallpaper

Offset	Length	Format	Contents
3314	1	Bool	Center frame wallpaper
3315	1	byte	Auto function keys (bit 0x01=Fnn, bit 0x02=PFnn, bit 0x04=Pann, bit 0x08=S/Fnn, bit 0x10=nn)
3316	1	byte	Auto function key text extract (bit 0x01=add text after '=', bit 0x02=show text only)
3317	1	Bool	Don't show 3D frame for frame wallpaper
3318	1	byte	<not used>
3319	2	Int	Saved toolbar X position
3321	2	Int	Saved toolbar Y position
3323	2	Int	Saved toolbar X extent
3325	2	Int	Saved toolbar Y extent
3327	2	Int	Saved keyboard bar X position
3329	2	Int	Saved keyboard bar Y position
3331	2	Int	Saved keyboard bar X extent
3333	2	Int	Saved keyboard bar Y extent
3335	24	Int[12]	Toolbar bitmap references 21-32
3359	24	Int[12]	Toolbar function references 21-32
3383	41	String[40]	Starting script name (SS*)
3424	2	Int	Saved function key bar X position
3426	2	Int	Saved function key bar Y position
3428	2	Int	Saved function key bar X extent
3430	2	Int	Saved function key bar Y extent
3432	1	Bool	Save toolbar position
3433	1	Bool	Use keyboard bar
3434	1	Bool	Use function key bar
3435	1	Bool	Do DKU printer tabbing

Offset	Length	Format	Contents
3436	1	Bool	Use secure sockets
3437	1	Bool	Verify secure server
3438	1	Bool	Verify client certificate
3439	4	DWORD	Secure protocol
3443	4	DWORD	Secure key exchange protocol
3447	13	String[12]	Client certificate name (SC*)
3460	1	Bool	Verify server name
3461	1	Bool	Verify against specific name
3462	13	String[12]	Name to verify against (SSC*)
3475	13	String[12]	Default FTP host (FH*)
3488	1	Byte	Dialog box font size (points)
3489	2	Int	Force high port number for TCP (value = lowest to assign)
3491	1	Byte	No copy options for hidden fields (bit 0x01=Clipboard, bit 0x02=Script, bit 0x04=API, bit 0x08=DDE)
3492	2	Bool	Font codepage
3494	1	Bool	DKU, Wincom compatibility
3495	1	Bool	TNVIP, don’t wait for turn
3496	1	Bool	Paste as block
3497	1	Bool	Include spaces in rectangles
3498	13	String[12]	SSH private key (PK*)
3511	13	String[12]	SSH password (PW*)
3524	1	Bool	Suppress Ggate coname list request
3525	1	Bool	Display 5250 errors in status bar
3526	1	Bool	Don't expand ENV in filenames
3527	1	Byte	Off centered graphics characters
3528	1	Bool	Fast graphic resize, no halftones

Offset	Length	Format	Contents
3529	41	String[40]	Default DDE name
3570	1	Bool	Keep insert mode on transmit (IBM only)
3571	1	Bool	Disable keyboard unlock type ahead
3572	1	Char	DKU RJF character
3573	1	Bool	No telnet timing mark
3574	1	Byte	Force SSH version (Default=0, V1=1, V2=2)
3575	5	String[4]	DGA local SCID
3580	1	Bool	not used
3581	1	Byte	DGA local DSA200 addr. 1
3582	1	Byte	DGA local DSA200 addr. 2
3583	1	Bool	not used
3584	1	Byte	DGA remote DSA200 addr 1
3585	1	Byte	DGA remote DSA200 addr 2
3586	1	Byte	DGA connect mode (Auto=0, TWAI=1, TWAA=2, TWSI=3, TWSA=4)
3587	16	Byte[16]	More configuration locks
3603	17	String[16]	SSHD server name
3620	1	Byte	Plink options (bit 0x01=hide plink, 0x02=user interactive, 0x04=password interactive)
3621	2	Word	Plink initial socket port
3623	1	Bool	Suppress SISO
3624	1	Bool	No ligation
3625	1	Bool	Arabic justify
3626	1	Byte	Arabic numerics (Default=0, Latin=1, Local=2)
3627	16	Rect	Keyboard layout position
3643	1	Bool	Extended char null

Offset	Length	Format	Contents
3644	1	Bool	Special 7800 wrap
3645	13	String[12]	QOS name
3658	1	Byte	<reserved>
3659	1	Bool	Don’t use themes
3660	11	String[10]	IBM printer message queue name
3671	11	String[10]	IBM printer message queue library
3682	1	Byte	IBM printer font
3683	1	Char	IBM printer form feed
3684	2	Word	IBM printer buffer size
3686	1	Bool	IBM printer transform
3687	15	String[14]	IBM printer name
3702	1	Byte	IBM printer paper 1
3703	1	Byte	IBM printer paper 2
3704	1	Byte	IBM printer envelope
3705	1	Bool	IBM printer Ascii899
3706	2	Word	IBM printer character set
3709	41	String[40]	Glink caption (GT*)
3750	1	Bool	Iconize to system tray
3751	1	Bool	Save to scrollback on XMT
3752	2	Word	Initial screen length
3754	2	Word	<reserved>
3756	1	Bool	UTF-8 encoding for host
3757	1	Bool	<reserved>
3758	17	String[16]	IBM 3270 print LU
3775	1	byte	Thai validation (0=basic,1=none, 2=strict)
3776	1	Bool	Thai no align

Offset	Length	Format	Contents
3777	1	Bool	Force old 3D background
3778	2	Word	IBM 3270 alternate row size
3780	2	Word	OIBM 3270 alternate column size
3782	2	Word	Button search left
3784	2	Word	Button search right
3786	2	Word	Button search top
3788	2	Word	Button search bottom
3790	1	Bool	Use two-tone background
3791	1	Bool	Enable SoftTerm commands
3792	32	Internal	Colors for two-tone background
3824	1	Bool	Full Unicode mode
3825	1	Bool	Use SSH buffering
3826	1	Bool	Don’t interpret VT MRC commands
3827	1	Bool	Change font when host resizes
3828	1	Bool	Check for upgrades

Example: Reading a byte

```
CRDB %1 47              * Kermit timeout value
MESSAGE ("Your Kermit timeout is " %1 " seconds.")
```

Example: Reading a byte with predefined meanings

```
CRDB %1 2               * parity setting
SHOW "Your parity setting is "
SWITCH %1
   CASE 0;  MESSAGE "7 bit even."
   CASE 1;  MESSAGE "8 bit none."
   CASE 2;  MESSAGE "7 bit odd."
   CASE 3;  MESSAGE "8 bit even."
   CASE 4;  MESSAGE "8 bit odd."
   DEFAULT; MESSAGE "Invalid!"
ENDSWITCH
```

Example: Reading a boolean value

```
CRDB %1 6              * initial roll mode setting
SHOW "Roll mode is initially "
SWITCH %1
   CASE 0;  MESSAGE "OFF."
   CASE 1;  MESSAGE "ON."
   DEFAULT; MESSAGE "Invalid!"
ENDSWITCH
```

Example: Reading a character

```
CREAD %1 1169 1        * Kermit packet header
FIX %1                 * fix up for display
MESSAGE ("Your Kermit packet header is " %1 ".")
```

Example: Reading an integer

```
CRDW %1 0              * two byte line speed
MESSAGE ("You are configured to run at " %1 " bps.")
```

Example: Reading a character array

```
CREAD %1 7 4           * answer to ENQ
MESSAGE ("You are set up to answer ENQ with '" %1 "'")
```

Example: Reading a string

```
CRDB %1 891 * download dir, length byte
CREAD   %2 892 24  * download dir, contents
SUBSTR  %2 %2 1 %1 * truncate to correct length
MESSAGE ("Your download directory is '" %2 "'")
```

Index

$

$$TERM.SCR 8

B

Built-in variables 13
Button bar 38

C

Caret 6
Case sensitivity 13
Command line
  parameters 13
Comments 7
Compiled scripts 7
Computations 12
  add 34
  calc 40
  divide 61
  multiply 116
  subtract 166
Concatenating strings 11
Configuration directory 14
Constants 6
Control characters 6
Conventions 31
Current directory 13

D

Debugging 4, 57, 150, 164
Delimiters 150
Demo directory 14
dial directory
  name 13
Dial directory
  calling 60
  comments 13
  entry number 13
  finding entries 59
  linking scripts 3
  manual dial 104
  marking 61
  modifying 54
  password 16
  phone number 16
  read an entry 64
  redial 137
  result code 14
  unmark entry 66
Direct execution 4
Disk space 14
Download directory 14, 64
Drag and drop 3

E

Editor name 14
Emulator mode 19
Emulator modes 111, 157
Encryption 7
Error level 14
Error message 14
Exponential format 12

F

File names
  extension 2, 3
File variables 20
Files
  appending 35

G

Glink directory 5, 15
GLWINOPT 3

H

Help texts
  button bar 39
Hexadecimal 6
Host initiation of scripts 3
Host message 15
Host name 15

I

Idle timer 15, 92, 154
Indirect variables 11, 20
Interfaces 52

L

Labels 4, 103
Limits 11, 12

M

Menu dialogs
  domenu 62
  fonts 110
  menu 109
  OK button 112
  operations 112, 113
  positioning 113, 114
  redisplay 140
  removal 171
  removing 118
  text lines 115
Menus
  addmenu 34
  buildmenu 37
  context menus 48
  delmenu 59
  disabling 106
  endbuild 68
  sepmenu 147
Multiple commands 6

N

Named variables 12, 58
Nesting scripts 7
NetBIOS 13
Notepad 1
Numeric variables 12

O

Object files 8, 119
Octal 6

P

Parameters 4, 14, 16, 18
Password 16
Patterns 19, 64, 66, 127

Q

Quotes 6

R

result code 14

S

Screen dump 65
Script directories 5
Script directory 5, 15
Script files
  general 1
Search rules 5
Sound 130
Starting a script 2
Startup script 3
Statistics 13
Status line 2

T

Terminate 4, 33, 91
Termination script 8
TN3270/5250 LU device 150

U

Upgrading scripts 8
Upload directory 17

User directory 15
User script directory 5, 15

V

Variables 11
Version testing 32

X

X.25 13