NAME

sqsh - Interactive database shell

SYNOPSIS

sqsh [*options***] [***args***...]**

[**−a** *count*]
[**−A** *packet_size*]
[**−b**]
[**−B**]
[**-c** *[cmdend]*]
[**-C** *sql*]
[**-d** *severity*]
[**-D** *database*]
[**-e**]
[**-E** *editor*]
[**-f** *severity*]
[**-h**]
[**-i** *filename*]
[**-H** *hostname*]
[**-I** *interfaces*]
[**-J** *charset*]
[**-k** *keywords*]
[**-l** *debug_flags*]
[**-L** *var=value*]
[**-m** *style*]
[**-o** *filename*]
[**-p**]
[**-P** *[password]*]
[**-r** *[sqshrc[:sqshrc ...]]*]
[**-s** *colsep*]
[**-S** *server*]
[**-t** *[filter]*]
[**-U** *username*]
[**-v**]
[**-w** *width*]
[**-X**]
[**-y** *directory*]
[**-z** *language*]

DESCRIPTION

Sqsh (pronounced skwish) is short for **SQ**shel**L** (pronounced s-q-shell), and is intended as a replacement for the venerable ‘isql’ program supplied by Sybase. It came about due to years of frustration of trying to do real work with a program that was never meant to perform real work.

Sqsh is much more than a nice prompt, it is intended to provide much of the functionality provided by a good shell, such as variables, aliasing, redirection, pipes, back-grounding, job control, history, command substitution, and dynamic configuration. Also, as a by-product of the design, it is remarkably easy to extend and add functionality.

OPTIONS

The following options may be used to adjust some of the behavior of *sqsh*, however a large portion of the configuration options are available only through environment variables which may be set at runtime or via a .sqshrc file.

Options may also be supplied in the **SQSH** environment variable. This variable is parsed prior to parsing the command line, so in most cases the command line will override the contents of the variable. Be aware that for options which are allowed to supplied multiple times, such as **-c**, supplying them both in a variable and on the command line will be the same as supplying them multiple times on the command line.

-a *count*	Sets the maximum *count* of failures (as determined by the **$thresh_fail** variable) that may occur before *sqsh* will abort. Setting this to 0 indicates that *sqsh* should not exit on errors. This value defaults to 0 and may also be set using the **$thresh_exit** variable. See section **EXIT STATUS** for details.
-A *packetsize*	Specifies the size of the network TDS packets used to communicate with the SQL server. This value must be between 512 and 2048, and be a multiple of 512. Check your SQL Server configuration to determine supported packet sizes. This value may also be specified at run-time using the **$packet_size** variable.
-b	Suppress the banner message upon startup. This is unnecessary in cases where stdout has been redirected to a file. This option may also be set via the **$banner** variable.
-B	Turns off all buffering of stdin, stdout, and stderr. This feature allows sqsh to be run from an interactive control script such as chat and expect.
-c [*cmdend***]**	Internally *sqsh* provides the command **\go** to send a batch of SQL to the database and provides a single alias, **go** for this command. Each time *cmdend* is supplied a new alias for **\go** is established.
-C *sql*	Causes the *sql* command to issued by sqsh, similar to the same behavior exhibited by the **-i** flag. This *sql* statment may not contain double quotes (this limitation will be lifted in future releases of sqsh).
-d *severity*	Sets the minimum SQL Server error severity that will be displayed to the user. The default is 0, and valid ranges are from 0 to 22. This may also be set using the **$thresh_display** variable. See section **EXIT STATUS**.
-D *database*	Causes *sqsh* to attempt to start with your database context set to *database* rather than your default database (usually master). This may also be set using the **$database** variable.
-e	Includes each command issued to *sqsh* to be included in the output. This option may also be set via the **$echo** variable (which is unrelated to the **\echo** command).
-E *editor*	Set the default editor to *editor*. This may also be set using the UNIX environment variable **$EDITOR** to the name of the editor desired.
-f *severity*	Sets the minimum *severity* level considered a failure by *sqsh*. This is the same as setting the **$thresh_fail** variable. See section **EXIT STATUS** for details.
-h	Turns off column headers and trailing "(# rows affected)" from batch output.
-i *filename*	Read all input from *filename* rather than from stdin.
-H *hostname*	Sets the client hostname as reported in sysprocesses. This may also be set via the **$hostname** variable.
-I *interfaces*	When a connection is established to the database, the *interfaces* file is used to turn the value of **$DSQUERY** into the hostname and port to which the connection will be made, by default this is located in **$SYBASE**/interfaces. This flag allows this default to be overridden.
-J *charset*	Specifies the character set to be used on the client side to communicate with SQL Server. This may also be set using the **$charset** environment variable.
-k *keywords*	Specifies a file containing a list of keywords to be used for keyword tab completion, if readline support has been compiled into *sqsh*. This file may also be set via the **$keyword_file** variable, which defaults to **$HOME/.sqsh_words**.
-l *debug_flags*	If *sqsh* has been compiled with -DDEBUG, this option may be used to turn on and off

debugging options. See the **$debug** variable, below.

-L *var***=***value* Sets the value of **$var** to *value*. This may be used to set the value of any *sqsh* variable even if an explicit command line variable is supplied for setting the variable. The **-L** flag may be used to set the value of non-configuration variables as well.

-m *style* Changes the current display style to *style*. Currently supported styles are **horiz**, **vert**, **bcp**, **html**, **meta**, **pretty** and **none**. The current display style may also be set using the **$style** variable or via the **-m** flag to the **\go** command.

-o *filename* Redirects all output to *filename* rather than stdout.

-p Display performance statistics upon completion of every SQL batch. This option may also be turned on via the **$statistics** variable, or by supplying the **-p** flag to the **\go** command.

-P [*password***]** The Sybase *password* for *username* required to connect to *server* (default, NULL). The *password* may also be set via **$password**. Supplying a password of ‘-’ causes the password to be read from the first line of stdin.

It should be noted that supplying your password on the command line is somewhat of a security hole, as any other user may be able to discover your password using **ps(1)**. It is recommended that your default password be stored in a .sqshrc file which is not readable by anyone other than yourself.

-r [*sqshrc[:sqshrc ...]***]**
Specifies an alternate *.sqshrc* file to be processed, rather than the default. If no *sqshrc* is supplied following **-r**, then no initialization files are processed. This flag **must** be the first argument supplied on the command line, all other instances will be ignored.

-s *colsep* Causes the string *colsep* to be used to delimit SQL column output columns, this defaults to " ".

-S *server* The name of the Sybase *server* to connect, the default of this is the external environment variable **$DSQUERY**. This value may also be set via the internal variable **$DSQUERY**.

-t [*filter***]** Enables filtering of command batches through an external program, *filter*, prior to being sent to the SQL Server. If *filter* is not supplied, then **$filter_prog** is used (default is ‘m4 -’). This value may also be set via the **$filter** and **$filter_prog** variables.

-U *username* The Sybase *username* to connect to the database as, this defaults to the username of the user running *sqsh*. The *username* may also be set via the **$username** variable.

-v Displays the version number, **$version,** and exits.

-w *width* The maximum output *width* of a displayed result set, this defaults to 80 (the maximum for this value is 256).

-X Initiates the login connection to the server with client-side password encryption (if supported). If either SQL Server does not recognize this option, or if the version of DB-Lib used to compile *sqsh* does not support this option, then it will be ignored. This option may also be set using the **$encryption** environment variable.

-y *directory* Specifies a SYBASE *directory* to use other than the value of **$SYBASE** in order to find the interfaces file.

-z *language* Specifies an alternate *language* to display *sqsh* prompts and messages. Without the **-z** flag, the server’s default language will be used. This may also be set using with the **$language** variable.

args If sqsh is run with the **-i** flag specifying an input file to be processed (rather then initiating an interactive session), arguments may be supplied on the command line to be passed to the input file. These arguments may be accessed using the variables **${0}**, **${1}**, ...

(see the **Variables** section, below, for more information).

INITIALIZATION

Upon startup, *sqsh* initializes all internal environment variables, commands, and aliases to their default values, it then looks in the system-wide configuration file (usually /usr/local/etc/sqshrc), followed by a local configuration file **$HOME**/.sqshrc (this may be overridden via the SQSHRC external environment variable). If this file is found it is executed just like a script would be using the **-i** flag.

The *.sqshrc* file may contain anything that could normally be typed at the prompt, however it should be noted that at the time this file is read *sqsh* has yet to establish a connection to the database, however most commands that perform database activity, such as **\go** will attempt to establish a database connection when executed (it may also prompt you for a password if necessary). Also, if database activity is required within this startup file, the **\connect** command (see **COMMANDS**, below) may be executed.

After the *.sqshrc* file has been executed, *sqsh* then parses any command line options (thus any variables set in your *.sqshrc* file may be overridden by command line options). Following that, if *sqsh* is run in interactive mode (i.e. without **-i** and if stdin is attached to a tty), it then looks for *.sqsh_history* and loads the contents of that file into this history buffer (see **BUFFERS**, below).

Immediately prior to establishing a connection to the database (either during startup, or by an explicit **\connect** or **\reconnect** command), the file **$HOME/.sqsh_session** is executed. The name of this file may be overridden using the **$session** variable.

COMMAND LINE

When a line is first read by *sqsh*, the first word is separated from the line. This word is then expanded of all variables (see **Variable Substitution**, below), followed by command expansion (see **Command Substitution**, below). The first word of the resulting string is then analyzed to see if it is either a valid *sqsh* command or alias.

The *sqsh* command line follows many of the same rules as Bourne shell, allowing file redirection, pipelining, command substitution, and backgrounding via the same syntax.

Comments

Any line beginning with a **#** following by a non-alphanumeric character (any character other than 0-9, a-z, A-Z, and _) causes the entire line to be ignored. Because of the possible collision with TSQL temp-table names, the line will not be ignored if the first character following the **#**, is alphanumeric.

Quoting

Quoting is used to prevent the interpretation of special keywords or characters to sqsh, such as white-space, variable expansion, or command substitution. There are three types of quoting, *escape*, *single-quotes*, and *double-quotes*.

Enclosing characters in single quotes preserves the literal interpretation of each character contained within the quotes. A single quote may not appear within single quotes, even when preceded by an escape. For example:

```
1> \echo I can not expand '$username'
```

outputs

```
I can not expand $username
```

The characters \\ are used to escape the meaning (and thus prevent the interpretation) of the character immediately following them. The \ character itself may be escaped. For example:

```
1> \echo I can\\'t expand '$username'
```

outputs

```
I can’t expand $username
```

The escape character may also be used to escape a new-line in order to perform a line continuation, in this case the new-line is discarded and the continued line is automatically appended to the previous line, for example:

```
1> \echo Hello \\
--> world!
Hello world!
```

Enclosing characters in double quotes preserves the literal meaning of all characters within them with the exception of **$**, **‘**, and \\. A double quote may be contained within double quotes by escaping it.

```
1> \echo "\\"I can’t deny it, I like $username\\", she said."
```

prints out

```
"I can’t deny it, I like gray", she said.
```

Expansion

After a line of input has been read, *sqsh* attempts to expand the line of any aliases (see **Aliasing**, below), following that it attempts to determine if the line begins with a command keyword. Once a line has been determined to contain a command name it has two types of expansion performed up it: *variable substitution*, followed by *command substitution* respectively. Following this expansion the command line is separated into words and the command is executed.

Variable Substitution

The character $ is used to indicate variable substitution or expansion within a word. These variables may be assigned values by the **\set** command like so:

```
1> \set name=value
```

name may be a character or underscore followed by any combination of characters, digits, or underscore, and may not contain any special characters, such as (’) and ("). The restriction on the first character being a digit is introduced because SQL allows the representation of money datatypes as $nn.nn where n is a digit.

value may contain anything, however if it is to include white-space, then it must be quoted (see **Words & Quoting**, above). Note that in order to prevent the expansion of a variable use either single quotes, or two \’s, like thus:

```
1> \echo \\$name
$name
```

Variables may be referenced in one of two ways:

$*variable* In this manner all characters, digits, and underscores are treated as the *name* of the variable until another type of character is reached (either a special character, or a white-space).

${*variable***}** The braces are required only when *variable* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. Note that the same effect may be achieved using double quotes.

It should be noted that because the variables are expanded prior to breaking the command line into words, if the contents of the variable contain white spaces, they are treated as significant by the parser. In the following example:

```
1> \set x="1 2 3"
```

```
2> \echo $x
```

the **\echo** command receives three arguments, "1", "2", and "3", although it looks as if only one argument was passed to it. This behavior is consistent with most shells (such as csh, bourne shell, etc.).

Command Substitution

Sqsh supports a second form of expansion called *command substitution*. This this form of expansion the output of an external UNIX command may be substituted on the command line. This expansion may be achieved by placing the command line to be executed in back-quotes (‘). For example:

```
1> \set password=‘/sybase/bin/getpwd $DSQUERY‘
1> \echo $password
ilikepickles
```

This this example, the external program **/sybase/bin/getpwd** is executed with the current contents of the $DSQUERY environment variable, the entire expression is then replaced with the output of **getpwd** (**ilikepickles**) prior to executing the **\set** command.

By default, the output of the substituted command is first broken into words according to the contents of the **$ifs** variable prior to assembling together back into the command line. So, by overriding the contents of **$ifs** you may affect the behavior of the substitution process. For example:

```
1> \set ifs=":"
1> \echo ‘echo hello:how:are:you‘
hello how are you
```

This mechanism is frequently useful for parsing input files, such as **/etc/passwd** into fields.

Input/Output Redirection

As with standard Bourne shell (and most other shells, for that matter), a command’s input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere on the command line, but only redirection that is specified *prior* to a pipe (|) actually has any effect on the behavior of internal *sqsh* commands (refer to **Pipes**, below).

<*word*	Use the file *word* as the standard input for the command. Typically very few *sqsh* commands actually read anything from stdin, so this will usually have no effect (see the **\loop** command).
[*n***]**>*word*	Associate the output of file descriptor *n* (stdout, by default) with file *word*. If this file does not exist it is created; otherwise it is truncated to zero length.
[*n***]**>>*word*	Append the the output of file descriptor *n* (stdout, by default) to file *word*, creating it if it does not exist.
[*m***]**>**&***n*	Redirect the output of file descriptor *m* (stdout by default), to same output as file descriptor *n*.

The order in which redirections are specified on the command line is significant, as the redirections are evaluated left-to-right. For example:

```
1> select * from select  /* Syntax error */
2> \go >/tmp/output 2>&1
1>
```

This statement first redirects the standard output of the **\go** command to the file /tmp/output, then redirects the stderr to the same file. So, when the commands fails, the error output will be found in the file /tmp/output.

However, by changing the order of redirection, you can completely change the meaning:

```
1> select * from select
2> \go 2>&1 >/tmp/output
Msg 156, Level 15, State 1
Server 'SQSH_TEST', Line 1
Incorrect syntax near the keyword 'select'.
```

In this case, error output will be sent to stdout, while what would have gone to stdout is redirected to /tmp/output (in this case /tmp/output will be empty).

Please read the section on **Background Jobs**, below, for detailed info on the interaction between file redirection and background jobs.

Pipes

A *pipeline* is a sequence of one or more commands separated by a '|', each command using the stdout of the preceding program for its own stdin. However the first command in the *pipeline* must be a *sqsh* command, and all other commands must be external (or UNIX) programs. Any *sqsh* command may be run through a pipeline, although for many of them (such as the **\set** command) it doesn't really make any sense to do this. The following is an example of a *pipeline*:

```
1> select * from syscolumns
2> \go | more
```

This command causes the result set generated by the **\go** command to be sent to the **more(1)** program, which then sends it to your screen, pausing at each screen full of data (this is the primary reason that I wrote *sqsh*).

There are several peculiarities in the way in which *sqsh* deals with *pipelines* as opposed to the way in which standard Bourne shell treats them.

Everything following the first occurrence of a pipe (|) character is broken into white-space delimited words, including such special shell commands as '2>&1' and other occurrences of pipes. If there are any variables contained in these words they are expanded following the same quoting rules as described in **Words & Quoting**, above, with the one exception that all quotes are left in place. These words are then re-assembled into a single string and shipped off to /bin/sh for processing.

In short, *sqsh* makes no attempt to interpret what follows the first pipe, instead it is shipped off to a "real" shell to do the work. The rationale behind this is that I was lazy and didn't feel like writing all of the same bizarre variable handling, &&'ing, ||'ing, grouping, and variable expansion rules that Bourne shell supports, and instead I let Bourne do the dirty work.

The advantage of this method is that you can do some very complex stuff after the pipeline, such as:

```
1> select * from syscolumns
2> \go | (cd /tmp; compress -c > syscolumsn.Z)
```

Not that I can think of any real reason to do this...but you can if you want to.

Background Jobs

Backgrounding provides a mechanism whereby you may run any *sqsh* command as a background process and continue working while it runs. *Sqsh* offers two types of backgrounding:

Deferred In this mode *sqsh* redirects all output of the background job to a temporary file (located in the directory **$tmp_dir**) while the job is running, so that the output is not intermixed with what you are currently working on. When the job completes you are notified of the process completion and the output may be viewed using the **\show** command.

Non-Deferred This corresponds to the common idea of a background process under UNIX. In this mode the output of the job is not implicitly redirected for you, and thus may become intermingled with what you are currently working.

The mode selection you choose is selectable via the **$defer_bg** variable (which defaults to ‘1’, or ‘On’). Typically the only reason to not use *deferred* mode is to prevent large result sets from filling up your file system.

To specify that a job be run in the background, simply append a & to the end of the command line, as:

```
1> sp_long_arduous_proc 1, 30
2> \go &
Job #1 running [xxxx]
1>
```

When *sqsh* encounters the & on the end of the command line it spawns a child process (with a Unix process id of *xxxx*) then the child process calls the **\go. \go** command then establishes a new connection to the database (using the current values of the **$DSQUERY, $username, $password** variables) and executes the shown query.

While the job is executing the commands **\jobs \wait** and **\kill** may be used to monitor or alter a currently running jobs (see section **COMMANDS**, below). When any jobs complete *sqsh* will display a notification, such as:

```
1> select count(*) from <return>
Job #1 complete (output pending)
2>
```

When a job completes, if it had no output, it is immediately considered terminated and will not show up in the current list of running jobs. However if the complete job has pending output, it will continue to be displayed as a running job (with the **\jobs** command) until a **\show** is used to display the output of the job.

There is a known bug with job backgrounding when used in conjunction with pipes, please refer to the **BUGS** section at the end of the manual.

BUFFERS

In normal **isql** only a two buffer are maintained; the buffer into which you are currently typing, and a buffer that contains the last batch executed (this is kept around for when you run ‘vi’, or ‘edit’).

Sqsh maintains several distinct sets of buffers:

Work Buffer This buffer corresponds directly to the **isql** work buffer. It is the buffer into which you enter the current batch prior to sending it to the database.

History Buffer This is actually a chain of 0 or more buffers (configurable by the **$histsize** variable) of the last **$histsize** batches that have been run. This buffer is only maintained when *sqsh* is run in interactive mode; that is, batches executed using the **-i** flag, or executed via redirection from the UNIX prompt will not be maintained in history (after all, they are already in a file somewhere).

If the variable **$histsave** is True (see section **SPECIAL VARIABLES**), and *sqsh* is in interactive mode, then the current history buffer is written to **$HOME**/.sqsh_history when the you exit. This file is then read back into *sqsh* the next time it is started in interactive mode.

Named Buffers At any time during a session the **Work Buffer**, or any of the **History Buffers** may by copied into a named buffer using the **\buf-copy** command (see section **COMMANDS**, below). These buffers are lost when you exit (however you may use the **\buf-save** command to save named buffers to a file).

Buffer Short-Hand

Many commands allow all of these buffers to be referenced in a short-hand fashion, very similar to the way that **csh(1)** references its commands history. Any of these shorthands may by used for any *buffer* parameter described in the **COMMANDS** section:

!.	The current work buffer.
!!	The last command executed (note, this is not available in non-interactive mode as it does not maintain a history).
!+	The next available history entry. This is a write-only buffer, so typically only applies to such commands as **\buf-copy.**
!*n*	Refers to history #*n*. Each time an entry is written to history it is assigned an increasing number from the last entry, with this short-hand you may reference any given history.
!*buf_name*	Just for consistency this is supplied as a reference to named buffer *buf_name*, however *buf_name* without the leading '!' is also considered correct.
buf_name	Refers to the named buffer *buf_name*.

Variables

Variables may also be contained within work buffers. Under these circumstances the variables remain unexpanded until the buffer is sent to the database (via the **\go** command), during which time they are expanded and replaced within the buffer. This behavior may be altered via the **$expand** variable (see **Special Variables**, below).

The following is an example of using variables within a buffer:

```
1> \set table_name=syscolumns
1> select count(*) from $table_name
2> \go
```

This is the equivalent of performing the query:

```
1> select count(*) from syscolumns
2> \go
```

directly. Typically this feature is useful for reusing large complex **where** clauses, or long column names.

Quoting rules apply the same in SQL buffers as they do in command lines. That is, any variables contained within double quotes (") are expanded and variables contained within single quotes (') are left untouched. Thus:

```
1> select "$username", '$username'
2> \go
```

yields the results

```

 ---- ---------
 gray $username

(1 row affected)
```

Command Substitution

As with the command line, the output of UNIX commands may also be substituted within a SQL buffer upon execution (once again, only if the **$expand** variable is set to 1, or true). In this circumstance the command contained with backquotes (‘) is replaced with its output prior to forwarding the buffer to SQL server. For example:

```
1> select count(*) from ‘echo syscolumns‘
2> \go
```

Causes the strings ‘echo syscolumns‘ to be replaced by the word *syscolumns* prior to executing the command. It should be noted that the contents of the substituted command are only executed at the time of the **\go** command, not when the line of SQL is input.

FLOW-OF-CONTROL

New with version 2.0 of sqsh, is the ability to perform basic flow-of-control and functions using the **\if, \while, \do,** and **\func** commands.

Blocks & SQL Buffers

All *sqsh* flow-of-control commands are block-based. That is, if the test expression of the command is met, then a block of sqsh-script will be executed. For example, the definition of the **\if** command is:

```
\if expression
  block
\fi
```

This *block* may be any number of lines of sqsh commands, SQL, or flow-of-control statements to be executed if the *expression* evaluates to a success condition (0).

Each *block* has its own SQL buffer for the duration that the *block* is executed. That is, the following statements:

```
1> /*
2> ** IMPROPER USAGE OF IF BLOCK
3> */
4> SELECT COUNT(*) FROM
5> \if [ $x -gt 10 ]
6>   sysobjects
7> \else
8>   sysindexes
9> \fi
4> go
```

will yield:

```
Msg 102, Level 15, State 1
Server ’bps_pro’, Line 1
Incorrect syntax near ’FROM’.
```

because the string ’sysobjects’ or ’sysindexes’ were inserted into their own SQL buffers. These buffers are discarded as soon as the end of the block was reached, and since a **\go** command was not contained within the block, no additional errors were generated.

Thus, the correct way to write the above expression would be:

```
1> /*
2> ** PROPER USAGE OF IF BLOCK
3> */
4> \if [ $x -gt 10 ]
5>   SELECT COUNT(*) FROM sysobjects
```

```
6>   go
7> \else
8>   SELECT COUNT(*) FROM sysindexes
9>   60
10> \fi
4> go
```

or, even:

```
1> /*
2> ** PROPER USAGE OF IF BLOCK
3> */
4> \if [ $x -gt 10 ]
5>   \set table_name=sysobjects
7> \else
5>   \set table_name=sysindexes
6> \fi
4> SELECT COUNT(*) FROM $table_name
5> go
```

Also, note that the line number displayed in the sqsh prompt resets to the current position in the outer SQL buffer after reaching the **\fi** terminator.

Expressions

All flow-of-control statements in sqsh take an *expression* to determine which *block* of code to execute. Just like UNIX's Bourne Shell, this *expression* is simply an operating system program that is executed by sqsh. If the command returns a success status (calls exit(0)), then it is considered successfull.

For example, with following statement:

```
\while test $x -lt 10
  block
\done
```

will execute the contents of *block* while the current value of $x is less than 10. Note that 'test' is a standard UNIX program to perform basic string or numeric comparisons (among other things). Also, unlike many shells, sqsh has no built-in version of 'test'.

Sqsh does, however, support the standard short form of 'test':

```
\while [ $x -lt 10 ]
  block
\done
```

With this expression the open brace ('[') is replaced by the sqsh parser with 'test', and the close brace (']') is discarded.

Unsupported Expressions

Currently sqsh does not support the standard shell predicate operators '&&' and '||'. These can be performed like so:

```
\if sh -c "cmd1 && cmd2"
  block
\done
```

\if statement

The **\if** command performs conditional execution of a sqsh *block* based upon the outcome of a supplied expression:

```
\if expr1
  block1
\elif expr2
  block2
\else
  block3
\fi
```

In this example, if expression *expr1* evaluates to true, then the block *block1* is evaluated. Otherwise, if the expression *expr2* evaluates to true, then block *block2* is evaluated. Finally, if all other tests fail *block3* is evaluated.

Note that, unlike Bourne Shell, every **\if** command must be accompanies by a trailing **\fi** statement. Also the sqsh parser is not terribly intelligent: The \else and \fi statements must be the only contents on the line in which they appear, and they may not be aliased to another name.

\while statement

The **\while** command executes a *block* of sqsh code for the while a supplied expression remains true.

```
\while expr
  block
\done
```

In this example, while the expression *expr* evaluates to true, then the block *block* is evaluated.

The **\break** statement may be used to break out of the inner-most **\while** or **\for** loop (more on **\for** below).

\for statement

The **\for** command executes a *block* of sqsh code for each *word* supplied:

```
\for var in word ...
  block
\done
```

For each *word* supplied, the value of the variable **$var** is set to the word and the *block* of code is executed. Execution ends when there are no more words in the list.

As with **\while** the **\break** statement may be used to break out of the inner-most execution loop.

\do command

The **\do** command is kind of a cross between a statement and a command. It is a form of **\go** (see below for details on the **\go** command) in which a *block* of sqsh code may be executed for each row of data returned from the query. When the *block* is executed, special sqsh variables #[0-9]+ (a hash followed by a number) may be used to reference the values in the returned query. For example the following command:

```
SELECT id, name FROM master..sysdatabases
\do
   \echo "Checkpointing database #2, dbid #1"
   use #2
   go
   checkpoint
   go
```

```
    \done
```

would cause a CHECKPOINT command to be issued in each database on the server.

Command line options

The **\do** command establishes a new connection to be used by the *block* of code when executed. By default, this connection is established to the current server (the current setting of **$DSQUERY**), using the current username (**$username**) and the current password (**$password**). This behavior may, however, be overridden using command line options:

-U *username*
Establishes the connection to the server as the supplied *username*.

-P *password*
Establishes the connection to the server using the supplied *password* (which is hopefully a valid password for the supplied *username*).

-S *server*
Establishes the connection to the supplied *server*.

-n Do not create a connection for use by the **\do** loop. This flag is mutually exclusive with the above flags. With this flag enabled, attempts to perform database commands within the *block* will generate a flurry of CT-Library errors.

Column variables

As mentioned above, the values of the columns in the current result set may be determined using the special #[0-9]+ variables. Thus, the variable #1 would contain the value of column number one of the current result set, and #122 could contain the value of the 122'nd column (column numbers begin at 1).

In the case of nested **\do** loops, values in previous nesting levels may be referred to by simply appending an addition '#' for each previous nesting level, like so:

```
SELECT id, name FROM sysobjects
\do
   SELECT indid, name FROM sysindexes WHERE id = #1
   \do
      \echo "Table ##2 (objid ##1) has index #2 (indid #1)"
   \done
\done
```

obviously, this isn't the way you would do this query in real life, but you get the idea.

When expanding columns with NULL values, the column variable will expand to an empty string (''). Also, references to non-existant columns, such as #0, will result in an empty string ('').

As with regular sqsh variables (those referenced with a '$'), column variables will not be expanded when contained within single quotes.

Aborting

If the **\break** or **\return** commands are issued during the processing of a **\do** loop, the current query will be canceled, the connection used by the loop will be closed (unless the **-n** flag was supplied) and the **\do** loop will abort.

\func command

The **\func** command is used to define a reusable block of *sqsh* code as a function. Functions are defined like so:

```
\func stats
   \if [ $# -ne 1 ]
      \echo "use: stats [on | off]"
      \return 1
   \fi
   set statistics io ${1}
   set statistics time ${1}
   go
\done
```

In this example a new function is established called *stats* that expects a single argument, either "on" or "off". Using this argument, *stats* will enable or disable time-based and I/O-based statistics.

Once established, the function may be called like so:

```
\call stats on
```

Causing all instances of ${1} to be replaced with the first command line argument to *stats*.

Command line options

Currently only one command line argument is available to the **\func** command.

-x Causes the function to be exported as a *sqsh* command. That is, the function may be invoked directly without requiring the **\call** command. This behavior is optional because command names can potentially conflict with T-SQL keywords. When using this flag it is recommended that you prepend a backslash (\) to your function name.

Function variables

As shown in the example above, several special variables are available for use within the body of the function. These are:

$# Expands to the number of arguments supplied to the function when invoked.

${0}..${*NN*}
Expands to positional arguments to the function. ${0} is the name of the function being invoked, ${1} is the first argument, ${2} the second and so-on, up to argument *NN*.

Note that, unlike most shells, *sqsh* requires that function arguments be referred to using the special curley brace syntax (${1}, rather than $1). The reason for this is that $1 is a valid MONEY value and using the curely braces gets rid of this ambiguity.

$? After the invokation of a function, this will contain its return value (see below).

Return value

A value may be returned from a function via the **\return** command. Like so:

```
\return N.
```

Where *N* is a positive value. This return value is available to the caller of the function via the **$?** variable. As convention, a return value of 0 is used to indicate a success.

If **\return** is not explicitly called, the default return value is the current value of the **$?** variable (which is set to 0 upon entry of the function). Thus, if any SQL statements are invoked within the function, the default return value of **$?** will be the last error code returned during the processing of the SQL statement.

COMMANDS

Read-Eval-Print

The read-eval-print loop is the heart of the *sqsh* system and is responsible for prompting a user for input and determining what should be done with it. Typically this loop is for internal use only, however they are open to the user because there are some creative things that can be done with them.

\loop [-i] [-n] [-e sql] [*file*]

The **\loop** command reads input either from a file, a supplied SQL statement, or from a user (see the options below), determining whether the current line is a portion of a TSQL statement or a **sqsh** command, and performing the appropriate action. When run in an interactive mode **\loop** is also responsible for displaying the current prompt (see **$prompt** below).

\loop completes when all input has been depleted (end-of-file is encountered) or when a command, such as **\exit** requests that **\loop** exit.

-i Normally, if *file* is supplied and does not exist, **\loop** will return with an error condition, usually causing sqsh to exit. By supplying the **-i** flag, control will be returned to the calling loop as if end-of-file had been reached (that is, with no error condition).

-n By default, **\loop** will automatically attempt to connect to the database if a connection has not already been established via the **\connect** command. The **-n** flag disables this behavior allowing **\loop** to process commands that do not require database support.

-e *sql* Causes **\loop** to process the contents of *sql* as if the user had typed it at the prompt and an implicit call to **\go** is automatically appended to the statement. If multiple instances of **-e** are supplied, they are all sent as a single batch to the SQL Server for processing. This option may not be used in combination with a *file* name as well.

file Specifies the name of a *file* to be used as input rather than reading input from the user or from the **-e** flag.

Database Access

Given the size and complexity of *sqsh* (just look at the length of this man page), it is amazing how few database manipulation commands that there actually are. The following are commands that affect or use the current database connection:

\connect [-c] [-D *db***] [-S** *srv***] [-U** *user***] [-P** *pass***] [-I** *ifile***]**

This command is used primarily for internal use to establish a connection to a database. If a connection is already established it has no effect, however if a connection has not been established and **$password** has not been supplied, then the password is requested and a connection is established. **\connect** Accepts the following parameters:

-c By default, the **\connect** command uses the contents of **$database** to determine the database context that should be used upon establishing the connection (this is used by **\reconnect** to preserve the current database context upon reconnection). The **-c** flag suppresses this behavior and the default database context of login is used instead.

-D *db*

Causes **\connect** to attempt to automatically switch the database context to *db* after establishing the connection. Using this flag is identical to setting the **$database** variable prior to establishing the connection.

-S *srv*

The name of the Sybase *server* to connect, this defaults to **$DSQUERY** if not supplied.

-U *user*

The Sybase *user* to connect to the database as, this defaults to **$username** variable if not supplied.

-P *pass*

The *password* for *user* required to connect to *server*. This defaults to **$password** if not supplied.

-I *ifile*
The full path of an alternate Sybase *interfaces* file to use.

\reconnect [-c] [-D *db***] [-S** *srv***] [-U** *user***] [-P** *pass***] [-I** *ifile***]**
The **\reconnect** command may be used to force a reconnection to the database using a new username, servername, or password (if desired). If this command fails, the current connection remains (if there is any), however if it succeeds then the current connection is closed and the new connection becomes the only active one.

All arguments that are accepted by **\connect** are also accepted by **\reconnect** (in fact **\reconnect** uses **\connect** to establish the new connection).

\go [options] [*xacts***]**
Sends the contents of the **Work Buffer** to the database, establishing a new connection to the database if one does not already exist (by calling the **\connect** above). It them displays the results of the query back to stdout and returns, causing the **Work Buffer** to be cleared and moved to the end of the **History Buffer**.

If the **Work Buffer** is empty and the **$repeat_batch** variable is set to "On", **\go** will attempt to re-run the last command executed (this will only work in interactive mode if history support is enabled).

\go accepts the following arguments:

-d *display*
If X11 support is compiled into *sqsh*, and X display mode is being used (see **-x**, below), then *display* will be used as the X display area for the result set. By default the environment variable **$DISPLAY** is assumed.

-f Turns off the display of the footer message "(%d rows affected)". Footer messages may also be turned off via the **$footers** variable.

-h Turns off all column headers. These may also be turned off via the **$headers** variable.

-m *style*
Temporarily changes the display style to *style* for the duration of the command. Currently supported styles are **horiz** (or **hor** or **horizontal**), **vert** (or **vertical**), **bcp**, **html**, **meta**, **pretty** and **none**. The display style may be permanently set via the **$style** variable or the **-m** command line flag.

-n Turns off variable expansion in the **Work Buffer** prior to sending it to the server, this may also be turned off via the **$expand** variable.

-p Turns on output of performance statistics when the result set has been successfully returned from the server. This may also be turned on via the **-p** command line argument to sqsh, or the **$statistics** variable.

-s *sec*
If the value of *xacts* is greater than 1, this causes *sqsh* to sleep for *sec* seconds before executing the next transaction. Note that the time spent sleeping is excluded from the statistical information displayed with the **-p** flag.

-t [*filter***]**
Filters the command batch through an external program, *filter*, prior to being sent to the SQL Server. If *filter* is not supplied, then **$filter_prog** is used (default is ‘m4 -’). This value may also be set via the **$filter** and **$filter_prog** variables.

-w *width*
Overrides the value of **$width** for the life of the query (see **$width** below).

-x [*xgeom***]**
Turns on the X11 display filter (only if X11 support is comiled into *sqsh*), which causes the

result set to be sent to a separate window. If *xgeom* is supplied, then this value will be used as **$xgeom** for the life of the query (see **$xgeom** below).

xacts Specifies number of times the contents of the *Work Buffer* should be executed. Note that, similar to isql, a result set will only be displayed during the final execution of the batch. Also, the contents of the *Work Buffer* are only expanded once, prior to the first execution, so the contents of the buffer will not change between subsequent executions.

\bcp [bcp_options] *table*

The **\bcp** commands acts as a sort of enhanced **\go** command that redirects the result set(s) of the batch to another server via the bcp protocol. While it is possible to **\bcp** the result set back to the current server (the **$DSQUERY** variable), this is achieved more easily via a SELECT INTO.

The nitty-gritty details of **\bcp** go like this: First the current SQL batch is expanded (unless the **$expand** variable is set to 0) and shipped off to the database for processing. If all goes well, a new connection is established to the destination database (as specified via $DSQUERY or the **-S** flag) to transfer the result set using bcp. Then, the output of the source database connection is bound to the new bcp connection and data transfer is performed.

\bcp can handle multiple result sets without any problem (including result sets returned from stored procedures, etc.) provided that all of the result sets are valid for the destination table.

The equivalent of a "bcp out" may be performed using the **bcp** display style setting and file redirection (see the **$style** variable).

-A *packet*
Specifies the TDS packet size used to communicate with the destination server. If not supplied this defaults to the value the **$packet_size** variable, or (if that is not set), the default server packet size (usually 512 bytes).

-b *batch_size*
The number of records transferred in a single transaction between servers. Note that reaching the end of a result causes the batch to be transferred, regardless of the value of *batch_size*. The default is the entire result set.

-I *ifile*
The full path of an alternate Sybase *interfaces* file to use.

-J *charset*
Specifies the default *charset* used to communicate with the SQL Server. This defaults to the current character set (the value of the **$charset** variable).

-m *maxerr*
The maximum number of batches that may fail before **\bcp** gives up the ghost (default is 10). Note, that this only refers to failures within a given batch. When performing a bcp of multiple result sets to a server, if a given result set has, say, too many columns or bad data types, then the entire bcp process is aborted regardless of the value of *maxerr*.

-N Indicates that the value for an identity column in the destination table is being supplied within the result set.

-P *pass*
The *password* for *user* required to connect to *server*. This defaults to **$password** if not supplied.

-S *serv*
The name of the Sybase *server* to connect, this defaults to **$DSQUERY** if not supplied.

-U *user*
The Sybase *user* to connect to the database as, this defaults to **$username** variable if not

supplied.

-X Causes password negotiation with the destination server to be performed using client-side encryption.

table As with regular **bcp**, *table* may be either a fully or partially specified table name in the destination server. Note, that since a new database connection is established during the bcp processes that the database context of the connection may not be the same as the current context, so it is usually safest to fully specify the table name in the form database.owner.table.

\rpc [rpc_opt] *rpc_name* **[[parm_opt] [***@var***=]***value* **...]**

The **\rpc** command is used to directly invoke a stored procedure call in the connected server. This command is particularly useful for communicating with an Open Server that does not directly support language calls.

\rpc invokes the remote procedure *rpc_name* with one or more parameters that may be named (using *@var*) or anonymous (by not supplying a name).

Unfortunately, due to the fact that Sybase's implementation of RPC's, does not directly support most implicit data type conversions (mainly between VARCHAR (the string you supply on the command line) and the most other data types (that the remote procedure is expecting), the syntax for the **\rpc** command is somewhat complex. However, in short here is how things work:

As the **\rpc** command line is being parsed, *sqsh* attempts to guess the data type of the parameter *value* based on the format (for example if it contains only digits, it is assumed to be an integer), *sqsh* then performs an explicit data type conversion prior to calling the remote procedure call. If *sqsh* guesses wrong, several flags are supplied to force it to perform the correct data type conversion (see **parm_opt**).

Display Options

The following options may be supplied anywhere on the command line and are used to affect the manner in which the result set(s) returning from the remote procedure call are displayed:

-d *display*
If X support is compiled into sqsh, the value of *display* is used as the X windows DISPLAY variable. Note, this is usually supplied with the **-x** flag, below.

-f Turns off the display of the footer message "(%d rows affected)". Footer messages may also be turned off via the **$footers** variable.

-h Turns off all column headers. These may also be turned off via the **$headers** variable.

-m *style*
Temporarily changes the display style to *style* for the duration of the command. Currently supported styles are **horiz** (or **hor** or **horizontal**), **vert** (or **vertical**), **bcp**, **html**, **meta**, **pretty** and **none**. The display style may be permanently set via the **$style** variable or the **-m** command line flag.

-w *width*
Temporarily sets the output width to *width*. The output width may be perminantly set via the **$width** varable.

-x [*xgeom***]**
Sends output to a separate X window. If *xgeom* is supplied, then the X window uses this geometry (see **$xgeom** for details).

Parameter Options

The following options may be supplied immediatly prior to specifying a parameter *value* and are used to affect the way in which *sqsh* interprets the contents of the *value* prior to calling the remote procedure. Although *sqsh* will allow any combination of these parameters to be combined, it only really makes sense to combine the **-x** flag with any other flag.

-b Indicates that the *value* that is specified should be converted to VARBINARY before calling *rpc_name*. This flag is implicit (i.e. you need not supply it) if *value* starts with "0x" and contains only digits.

-c Indicates that the *value* that is specified should be converted to VARCHAR prior to calling *rpc_name*. This flag is implicit if *value* does not match any of the implicit conversions for the other data types.

-d Indicates that the *value* that is specified should be converted to double (float) before calling *rpc_name*. This flag is implicit if *value* is in valid floating point notation (e.g, 0.1, .1, 1.4e10, or 4e10).

-i Indicates that the *value* that is specified should be converted to integer (int) before calling *rpc_name*. This flag is implicit if *value* contains only digits (and, optionally, a leading sign).

-y Indicates that the *value* that is specified should be converted to money before calling *rpc_name*. This flag is implicit if *value* begins with a "$", and contains only digits and, optionally, a decimal.

-n Indicates that the *value* that is specified should be converted to numeric before calling *rpc_name*. This flag is never implicit, as *value* would always match either int (**-i**) or float (**-d**); however, both of these types will implicitly be converted to a numeric as necessary by the procedure call.

-u Indicates that *value* should be ignored and treated as a NULL value, This flag is implicit if *value* is "".

Buffers

The following commands may be used to create, destroy, or manipulate the various buffers described in the **BUFFERS** section, above.

\reset

The **\reset** command corresponds directly to the *isql* ‘reset’ command, returning a request to the read-eval-print loop to clear the contents of the current **Work Buffer** and, if you are running in interactive mode, place a copy of the buffer into the **History Buffer**. The alias **reset** is automatically established upon start-up of *sqsh* for backward compatibility with *isql*.

\redraw

Returns a request back to the current read-eval-print loop for it to redisplay the current **Work Buffer**. If run from non-interactive mode, this command has no effect.

\history

Displays the last **$histsize** batches that have either been sent to the database via the **\go** command or cleared from the **Work Buffer** via the **\reset** command.

\buf−copy *dst−buffer* **[***src−buffer***]**

Copies the contents of *src-buffer* (defaults to **!.**, the **Work Buffer**, if not supplied), to *dst−buffer*. Refer to **BUFFERS** for information on buffer naming conventions.

\buf−get *buffer*

The **\buf−get** command is supplied as a shorthand method of running **\buf−copy** It is the equivalent of running:

```
\buf-append !. buffer
```

\buf−append *dst−buffer* **[***src−buffer***]**

Appends the contents of *src−buffer* (defaults to !.) to the contents of *dst−buffer*, if it exists. If *dst-*

buffer doesn't exist it is created.

\buf−save [-a] *filename* **[***src−buffer***]**
Saves the contents of *src−buffer* (defaults to !.) to *filename*. If the **-a** flag is supplied the contents are appended to *filename* rather than overwriting the current contents.

\buf−load [-a] *filename* **[***dst−buffer***]**
Copies the contents of *filename* in *dst−buffer* (defaults to !.). If the **-a** flag is supplied, the contents of *filename* are appended to *dst-buffer*. Note that it is illegal to attempt to write to the contents of the history buffer.

\buf−show [*buffer*]
Displays the contents of the named *buffer*. If *buffer* is not supplied, then the contents of all named buffers are displayed. This command is slightly different from the commands above in that it is only legal to supply a **Named Buffer** *buffer*, **History Buffers**, and the **Work Buffer** will have no results.

\buf−edit [-r *read−buf*] [-w *write−buf*]
The **\buf−edit** command is used to edit the contents of a buffer and place the changes into another buffer. This command may only be run while in interactive mode. If *read−buf* is not supplied then the buffer to be edited defaults to !., if it is not empty, otherwise it defaults to !!. If *write−buf* is not supplied then the edited buffer is written back to !..

By default, **\buf−edit** uses the environment variable **$EDITOR** first, followed by **$VISUAL** to determine which editor to use, defaulting to 'vi' if the variable is not set.

It is important to note that as of release 1.2, **\buf−edit** is no longer able to use the name of an alias to it as the name of the editor to launch. This is primarily due to the change in the behavior of alias' (see section **Aliasing**, below, for details).

The commands **edit vi** and **emacs** are automatically established upon startup of *sqsh* for backward compatibility with *isql*.

Variables

The following command(s) are used to manipulate the contents of internal variables and environment variables. There aren't many of them right now, but there may be more in the future.

\set [-x] [*name***=***value* **...]**
If no arguments are supplied to **\set** then the current values of all variables are displayed. Otherwise the variable *name* is set to *value*. Note that some internal variables (see **SPECIAL VARIABLES**) may only be set with certain *value*s, so this action may fail, leaving the previous contents on *name* in tact. The **-x** flag causes the variable to be exported to the environment of any programs launched from *sqsh*.

Job Control

The following commands are used to view the status of, or manipulate background jobs that are currently running, these correspond roughly to the commands supplied by such shells as **csh(1)**.

\jobs Displays the status of any currently running jobs, including whether or not these jobs have pending output, how long they have been running, and when they were started.

\wait [*job_id***]**
Will pause until job designated by *job_id* completes. If *job_id* is a negative number then **\wait** will pause until *any* pending jobs completes. If there are no jobs pending, or *job_id* does not belong to a running job, then an error message is displayed.

\kill *job_id*
Terminates the job specified by *job_id*, throwing away any output that may be deferred for the job. If *job_id* is not a running job then an error message is displayed.

\show *job_id*

Displays the deferred output of completed background job *job_id* and removes the job from the list of pending jobs (removing the defer file in the process). If *job_id* is still running, or is not a valid complete job, then an error message is displayed.

Aliasing

As of release 1.2, *sqsh* supports full *csh*-style command aliasing. With this feature, *sqsh* checks the first word of each line, *prior to any form of expansion*, to see if it matches the name of an existing alias. If it does, the command is reprocessed with the alias definition replacing its name. Unlike *csh*, however, only one form of history substitution is available within an alias: the '**!**∗' entry, indicating the current line being expanded. If no history expansion is called for, the arguments on the command line remain unchanged.

Like *csh*, aliases are not recursively expanded, so it is perfectly legal to create an alias that expands to a command by the same name.

The following command is used to create an alias:

\alias [*alias_name*=*alias_body***]**

If no arguments are supplied to the **\alias** command, then the list of aliases currently in effect is displayed. Otherwise, it creates a new alias with a name of *alias_name* and a body of *alias_body*; if *alias_name* already exists the body of the existing *alias_name* is replaced with the new definition.

After defining the new alias, whenever *sqsh* encounters a line beginning with *alias_name*, the remainder of the line is replaced with *alias_body* before any further processing is performed.

If the string '**!**∗' exists anywhere within *alias_body*, the arguments supplied to the alias are inserted at that point, otherwise the argument are appended to the end of the alias definition. For example:

```
1> \alias hi='\echo !∗ said hello'
1> hi Scott
Scott said hello
```

where as if the alias does not include the **!**∗ keyword, then it behaves like so:

```
1> \alias hi='\echo said hello'
1> hi Scott
said hello Scott
```

It is perfectly legal to include a **!**∗ more than once within a given *alias_body*. Currently there is no way to escape the string **!**∗, if you really need this feature send me mail.

\unalias *alias_name*

Removes *alias_name*.

Miscellaneous

The left over commands.

\exit The **\exit** command requests that current read-eval-print loop cease processing. When the last loop returns, *sqsh* **exit(1)**s.

\abort Causes all nested read-eval-print loops to abort processing, causing *sqsh* to exit with an exit value of 254 (see section **EXIT STATUS**).

\read [-a] [-n] [-h] *var_name*

Reads a line of input from the user, placing the text of the line in the variable *var_name*. If the **-n** is used, then the trailing new-line is left on the line of text, and if **-a** is supplied, then the text of the line is appended to the existing value of *var_name*. The **-h** flag turns off echoing of typed characters back to the user.

\sleep *seconds*

Causes *sqsh* too pause for *seconds*. This is useful within scripts of batches need to need to pause briefly between batches (it was primarily useful to me for testing background jobs).

\echo [-n] [*args* **...]**

Just like the UNIX **echo(1)**, this prints its arguments to stdout, followed by a new-line. If the **-n** flag is supplied, the new-line is omitted.

\warranty

Displays the standard GNU warranty.

\help [*command***]**

Without any arguments **\command** displays a brief list of all available commands, otherwise, it provides specific help for *command*, if available. When help is requested on a specific *command*, **\help** looks for the file **$help_dir/***command***.hlp** and displays it to stdout.

\shell [*shell command***]**

If *shell command* is not supplied then *sqsh* executes **$SHELL.** If the **$SHELL** variable has not been set, then, by default, /bin/sh is executed. Otherwise, if *shell command* is supplied then it is executed. The exit status of the command executed is stored in the special **$?** read-only environment variable.

\lock Locks the current session until the correct password is typed. By default **\lock** attempts to use the UNIX password (from /etc/passwd) associated with the user running *sqsh*, however if the **$lock** variable is set then the contents of that is used for validation instead.

Note, on systems using Shadow Passwords (in which even the encypted password is unavailable), **\lock** will only work using the **$lock** variable.

Aliases

The following aliases are established upon startup of *sqsh*, and are provided primarily for backward compatibility with **isql**. These may be removed at any time using the **\unalias** command (either at the prompt, or within your .sqshrc file).

! The **!** alias is provided as a **csh(1)**-like history mechanism, and is an alias of **\buf−append.** With release 0.7, this alias is provided only for backwards compatibility with previous releases of *sqsh*. See **SPECIAL VARIABLES**, **$history_shorthand** for details on the new shorthand mechanism (the new shorthand more closely resembles that of **csh**).

reset An alias for the **\reset** command, which causes the contents of the current work buffer to be cleared and copied to history (if in interactive mode).

exit and **quit**

An alias for the **\exit** command, causes the current read-eval-print loop to complete.

edit, **vi**, and **emacs**

These are provided as aliases for the **\buf−edit** command. See **COMMANDS-Buffers** for information on the interactions between **\buf−edit** and aliases.

go Provided as an alias for the **\go** command (for obvious reasons).

help An alias for the **\help** command.

In−Line \go

If the variable **$semicolon_hack** is set to 1 (on), then sqsh supports what is called an in−line **\go** feature. This allows the current command batch to be terminated and sent to the database in a single step by appending a ‘;’ onto the end of the current work buffer. This allows

```
1> sp_who;
```

To behave in the same manner as if you had typed:

```
1> sp_who
2> \go
```

Likewise, anything following the semicolon is passed to the **\go** command just as if it was run as a normal command:

```
1> sp_who ; 2>/dev/null | more
```

Unlike most other **isql** replacements, *sqsh* attempts to be smart about the semicolons. If a semicolon is contained within a set of single or double quotes it will not be interpreted. This includes multiple quotes. For example:

```
1> select "This is a multiple line
2> quote; it is smart!" ;
```

In the above example, only the second semicolon (the one at the end of the line) will be interpreted.

SPECIAL VARIABLES

There are several options that are configurable via the command line options to *sqsh*, however these are by no means complete. There are many aspects of *sqsh*'s behavior that may only be modified by setting special variables (in fact, the command line options really only set these variables for you).

Variable Datatypes

Next to all of the variables that follow is the type of data with which they may be set. Any attempts to set the variable with a type of data that it does not accept will fail.

string	Any sequence characters.
boolean	A positive *boolean* value may be represented as either "True", "Yes", "1", or "On" (case insensitive) and a negative boolean value may be represented as "False", "No", "0", or "Off" (case insensitive). However, internally the value of the variable will always be represented as either a "1" or "0".
path	Must be the *path* name that is readable by the *sqsh* program.
int	Must be one or more digits. Note that some variables also restrict the range of the integer.
date-spec	This is a string of the format used to specify dates and times for the **date(1)** command, or the **strftime(3C)** and **cftime(3C)** standard C library functions. For example '%H:%M:%S' specifies a time of hours in 24 hour format, followed by a colon, followed by minutes, followed by a colon, followed by seconds.
float-format	A string of the format *pp.ss*, where *pp* is the total precision of a floating point value (the total number of digits to be displayed, including those following the decimal) and *ss* is the scale of the value (the total number of digits following the decimal to be displayed).

Variables

The following variables have special meanings within *sqsh* and the setting of these variables alter the behavior of the shell.

$? **(int)**
This variable may contain the following return value:

- The most recent error number return from the SQL Server (@@errno) of severity > 10 (above informational messages).
- The exit value of a previously executed pipe command.
- The return value of the most recently executed sqsh function.

$# **(int)**
Contains the number of arguments passed into the sqsh function or script.

${0}..${NN} **(int)** Used to reference positional function arguments. Argument ${0} is the number of the function being called, ${1} is the first argument, etc.

autouse **(string)** Note: the meaning of this variable has been deprecated. If **$autouse** is set, and the **$database** variable has not been set, then this variable causes **\connect** to perform a "use **$autouse**" once a connection has been established. This variable may also be set using the **-D** command line option.

banner **(boolean)** Turns off the banner message displayed on startup, this variable defaults to 1 and may also be turned off using the **-b** command line argument.

batch_failcount **(int)**
This internal variable is used to keep track of the number of batches that have failed to execute (essentially, the number of times that the error handler was called). A batch is considered failed whenever an error of severity **$thresh_fail** is encountered. When **$batch_failcount** reaches **$thresh_exit** *sqsh* exits with an exit value of the total number of batches that have failed. Setting **$batch_failcount** to the string "" will cause it to reset to zero, any other value may have unpredictable results. See **EXIT STATUS** for details.

batch_pause **(boolean)**
Causes a "Paused. Hit enter to continue..." message to be displayed after each batch is executed. This variable, in conjunction with **$echo** is good for debugging SQL scripts.

bcp_colsep **(string)**
Used as a separator between columns during BCP style output (see the **$style** configuration variable and the **-m** option to the **\go** command). The default setting is "|".

bcp_rowsep **(string)**
Used as a separator between rows during BCP style output (see the **$style** configuration variable and the **-m** option to the **\go** command). Note that, a newline ("\n") is automatically appended this this value and should not be supplied.. The default setting is "|".

bcp_trim **(boolean)**
Controls whether or not BCP style output trims trailing spaces from fixed length columns. The default is "True".

charset **(string)** If this variable is set prior to establishing a connection with SQL Server, then during the connection *sqsh* will request that the server transform to and from the requested *charset*. After establishing a connection, this variable is automatically set to the current character set in use.

clear_on_fail **(boolean)**
Normally, whenever the **\go** command is run, *sqsh* clears the current work buffer of its contents, moving them to history. Setting **$clear_on_fail** to 0, leaves the current work buffer in-tact if a failure is encountered while sending the contents to the database. The default value is 1, or on.

colsep **(string)** Causes the string *colsep* to be used to delimit SQL column output columns, this defaults to " ", it may also be set via the command line argument **-s**.

colwidth **(int)** Used to control the maximum column width displayed by the **pretty** display style (see **$style** below). If a row of a column exceeds this width, it will be wrapped in a relatively visually appealing manner at **$colwidth** characters. Note, however, that if there is enough screen width to hold all columns **$colwidth** may be exceeded until the width of the screen is reached.

date **(date-spec)** This variable may be set with a date format (see the man page for **date(1)**), and the variable expands to the current date in the supplied format. The default format for this variable is %d-%b-%y (e.g. 02-Feb-1996).

datetime **(date-spec)**

This variable may be set with a date format similar to **$date** and **$time** and is used to control the display format of all SQL Server DATETIME and SMALLDATETIME columns.

Note that this features relies upon the operating system specific locale information for determining such things as the name of the month and day, rather than going through the CT-Lib locale information. This means that the date format could potentially miss-match the locale as requested using the **-z** flag. For example, if *sqsh* is run on an operating system configure for US English, but requests French as the language of choice using **-z**, the use of **$datetime** will cause all date information to be displayed in US English rather than French.

Ordinary characters defined in the variable are left in place without any conversion. Characters introducted by a '%' character are replaced during display of a column value as follows:

[] Any contained between a pair of braces ('[' and ']') will be removed when displaying SMALLDATETIME columns. This feature is particularly useful for removing the seconds and milliseconds values which are not applicable to SMALLDATETIME anyway. For DATETIME columns, only the actual braces will be removed.

%a The abbreviated weekday name according to the current operating system locale.

%A The full weekday name according to the current operating system locale.

%b The abbreviated month name according to the current operating system locale.

%B The full month name according to the current operating system locale.

%c The preferred date and time representation for the current operating system's locale.

%d The day of the month as a decimal number (range 0 to 31).

%D The date in US format (mm/dd/yy).

%H The hour as a decimal number using a 24-hour clock (range 00 to 23)

%I The hour as a decimal number using a 12-hour clock (range 01 to 21)

%j The day of the year as a decimal number (range 001 to 366).

%m The month as a decimal number (range 10 to 12).

%M The minute as a decimal number.

%p Either 'am' or 'pm' according to the given time value, or the corresponding strings for the current operating system locale.

%r The time in 12-hour format (hh:mm:ss [AM|PM]).

%s Seconds since the epoc (1970-01-01 00:00:00 UTC) (this is not supported on all systems).

%S The second as a decimal number.

%T The current time in 24-hour format (hh:mm:ss).

%u The millisecond as a decimal number.

%U The week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week.

%W The week number of the current year as a decimal number, starting with the first Monday as the first day of the first week.

%w The day of the week as a decimal, Sunday being 0.

%x The preferred date representation for the current locale without the time.

%X The preferred time representation for the current locale without the date.

%y The year as a decimal number without a century (range 00 to 99).

%Y The year as a decimal number including the century.

%Z The time zone (e.g., EDT), or nothing if not time zone is determinable.

%% A literal '%' character.

database **(string)** If this variable is set prior to establishing a connection to the SQL Server, the a "use **$database**" is performed immediately after the connection is established. Once a connection has been established this variable will automatically be set to the current database context.

debug **(string)** If *sqsh* has been compiled with debugging enabled (-DDEBUG), this variable may be used to control the amount of debugging output displayed. **$debug** may be set to a pipe (|) delimited (logical OR) set of the following words to turn on various pieces of debugging: **FD**, **SIGCHLD**, **ENV**, **JOB**, **AVL**, or **ALL**.

defer_bg **(boolean)**
Normally, when a job is run the in the background (via a '&' on the command line), the output of the job is deferred to a temporary file (located in $*tmp_dir*) until the user requests the output to be displayed. This way the results of the job will not interfere with what the user is going. Setting this variable

echo **(boolean)** Setting **$echo** to on (1) causes each command submitted to the database via the **\go** command to be displayed prior to the output. This variable defaults to 0 (or off), and may also be set using the **-e** command line option.

encryption **(boolean)**
Setting the **$encryption** variable prior to establishing a connection to the server will cause the login connection to be initiated using client-side password encryption. This variable may also be set using the **-X** command line option.

exit_failcount **(boolean)**
Settings this value to 1 causes *sqsh* to return an exit status of **$batch_failcount** rather than 0, upon a non-error termination. See **EXIT STATUS** for details. The default value is 0.

expand **(boolean)** Be default when the **\go** command is executed the contents of the current work buffer is expanded of all environment variables prior to being sent to the database for execution. By setting this variable to "0", the buffer will no longer be expanded before being sent to the database. This is useful when you either (1) have strings in the buffer that contain a '$' and you don't want them to be expanded, or (2) for performance reasons; it takes time (and an extra copy of the buffer) to perform the variable expansion.

filter **(boolean)** Toggles the filtering the SQL batch through an external program (defined by the **$filter_prog** variable, below) prior to being sent to the SQL Server. Default is '0', or 'off'.

filter_prog **(string)**
Defines the external program through which the SQL batch will be filtered prior to being sent to the SQL Server. This variable is ignored if **$filter** is set to '0' or 'off'. The default is 'm4 -'.

float **(float-format)**
Defines the display format (the precision and scale) for all floating point values

displayed by sqsh. The default is ‘18.6’. Note that values exceeding the defined precision are not truncated, so setting this value too low may cause columns in a result set to be miss-aligned.

footers **(boolean)** Toggles the "(%d rows affected)" following a result set. The default for this variable is ‘1’.

headers **(boolean)**
Toggles the column headers preceding a result set. The default for this variable is ‘1’.

help_dir **(path)** This is the location of the help files used by the **\help** command, typically it defaults to something like /usr/local/lib/sqsh/help.

histnum **(int)** Contains the history number that will be assigned to the current command batch as soon as the **\go** command is executed. This variable should be considered read-only.

history **(path)** This is the location of the history file used to store and retrieve a users history during start-up and shut-down. This defaults to **$HOME**/.sqsh_history. This variable is expanded each time it is referenced by sqsh, much in the same way that **$prompt** is each time the prompt is displayed.

history_shorthand **(boolean)**
This variable is only meaningful within an interactive session. If set, it turns on the ability to append any named buffer or history buffer onto the current work buffer in a ‘sh’ history style, such as ‘!40’. Be careful with this feature, *sqsh* is not terribly intelligent with looking for history shorthand, so it is possible that it may get confused (although, it is smart enough to ignore !’s in quoted strings).

histsave **(boolean)**
The value of this variable is used by *sqsh* to indicate whether the history should be save to **$history** prior to shutdown.

histsize **(int)** The value of this variable is used to alter the maximum number of history entries are are maintained by *sqsh* (the default is 10). Note that decreasing the value of this variable causes some history entries to be lost.

hostname **(string)**
Used during the connection process to indicate to SQL Server the name of the host from which *sqsh* is connecting. This variable may also be set using the **-H** flag.

interactive **(boolean)**
This is a variable used internally and should probably not be altered by the user. If **$interactive** is ‘0’, then the prompt is not displayed, the history is neither read nor written and some user messages are suppressed.

interfaces **(path)** This is the full path name of the interfaces file, it defaults to **$SYBASE**/interfaces.

keyword_completion **(int/string)**
This variable only applies if GNU Readline support has been compiled into *sqsh*. **$keyword_completion** is used to control the TSQL keyword completion feature in readline, and may be set using either an integer between 0 and 4, or one of the strings *none*, *lower*, *upper*, *smart*, or *exact*. If it is set to either 0 or *none*, then no keyword completion is performed (this is the default). *lower* or 1, causes *sqsh* to complete the keyword in lowercase, regardless of the case that the partially completed keyword was typed. *upper* or 2 forces completion to be performed in upper case, *smart*, or 3, basis the decision on case upon the first character of the partial keyword, and *exact* completes the keyword in exactly the same case as defined in the **case).**

keyword_file **(string)**
If readline support has been compiled into *sqsh*, and *sqsh* is being run in interactive mode, the contents of this file are used for keyword tab completion by readline rather

than the default set of TSQL syntactical keywords. The default is **$HOME/.sqsh_words**.

language **(string)** The **$language** variable is used while establishing a connection to the server to specify the national language used to display system prompts and messages. The variable will automatically track the current language setting of the server. This may also be set via the **-z** flag.

lineno **(int)** This is an internal variable and should not be altered by the user. It is used to maintain the line number that is being typed into within the current work buffer.

linesep **(string)** Used to configure the line separator for the horizontal display style, this defaults to "\n\t".

lock **(string/write-only)**
Defines the password to be used by the **\lock** command. If unset or set to the string "NULL", then the UNIX password of the user running *sqsh* is used instead. Note that **$lock** will always expand to the string "*lock*" if referenced.

newline_go **(boolean)**
This flag is used as a horrible kludge to support an "empty" alias for the **\go** command, that is, the equivalent of supplying "-c ’’" on the command line. When on, an empty line is interpreted as a call to the **\go** command. This feature is not recommended but is supplied for completeness.

maxlen **(int)** Controls the maximum amount of data that will be displayed (in any display mode) in a single column. This setting will automatically truncate the output of particularly large datatypes (such as TEXT) to the value supplied. The default setting is 8192 bytes (8KB).

output_parms **(boolean)**
Flag used to enable to disable the display of output parameter result sets from stored procedures. The default is to enable the display.

packet_size **(int)** Defines the size of the TDS packets used to communicate with SQL. Changing the value of the variable will not affect the current connection but will take effect upon the next **\reconnect** command. Specifying a value of NULL indicates that the default packet size is desired.

password **(string/write-only)**
This is the users current password. A NULL password may be assigned using an explicit "NULL" string. For security reasons, when referenced the **$password** variable will always expand to the string "*password*".

prompt **(string)** This variable is used by *sqsh* to build your current prompt. Any variables contain within **$prompt** are expanded each time the prompt is displayed. The default value for this is ‘${lineno}> ’.

prompt2 **(string)** This contents of this prompt are expanded and displayed during interactive use when *sqsh* requires additional input, such as during a line continuation. The default value is ‘--> ’.

rcfile **(path)** Contains a colon (:) delimited list of sqsh resource (sqshrc) files. The default setting is /usr/local/etc/sqshrc (unless overridden by the --prefix option when sqsh was compiled) followed by **$HOME**/.sqshrc).

readline_history **(string)**
If readline support has been compiled into *sqsh*, the contents of the readline line-by-line history will be written to the file specified by the **$readline_history** variable. The default is **$HOME/.sqsh_readline**.

readline_histsize **(int)**
If readline support has been compiled into *sqsh*, the value of **$readline_histsize** specifies

the number of lines that are saved in the readline line-by-line history. Setting this to a value of 0 causes every line to be saved. The default value is 100.

real **(float-format)**
Defines the display format (the precision and scale) for all real values displayed by sqsh. The default is '18.6'. Note that values exceeding the defined precision are not truncated, so setting this value too low may cause columns in a result set to be miss-aligned.

repeat_batch **(boolean)**
When set to **On** or **True**, a **\go** executed with an empty **SQL Buffer** will cause the previous batch to be re-executed.

script **(string)** If *sqsh* is run using the **-i** flag, then this variable contains the name of the script being executed.

statistics **(boolean)**
Setting **$statistics** to 1 causes timing statistics to be displayed upon the successful execution of every batch of SQL. This variable may also be set via the **-t** command line flag, or by supplying **-t** to the **\go** command. **$statistics** defaults to 0.

semicolon_cmd **(string)**
When **$semicolon_hack** (see below) is enabled, this contents of this variable is executed when a semicolon is encountered in the **SQL Buffer**. This variable defaults to the string '**\go**'.

semicolon_hack **(boolean)**
Toggles on the ability to use a ';' as an in−line command terminator. This feature is not recommended and is only in here because enough users complained. See section **COMMANDS, In-Line Go**.

SHELL **(string)** The name of the shell to be used to execute pipes and to be used by the **\shell** command (default '/bin/sh').

style **(string)** Selects result set display style. Currently six styles are supported. The **horiz** (which may also be defined as **hor** or **horizontal**), closely resembles the output of isql, with the traditional columnar output.

The **vert** (or **vertical**) style rotates the output, so that every line is represented by a column name followed by a column value. This is nice for looking at particularly wide output.

The **bcp** style displays results in a format amenable to bcp'ing the result set back into another table. That is, every column value is separated by **$bcp_colsep** with the final column separated by **$bcp_rowsep** followed by a newline (\n). If **$bcp_colsep** or **$bcp_rowsep** are not defined then '|' is used as the default separator. Note that this output does not work well with COMPUTE columns, and uses the default conversion methods for all data types (that is, **datetime** columns may truncate the millisecond).

The **html** display style outputs all result sets in the form of an HTML <TABLE> construct. This mode is ideal for the use of sqsh as a CGI application.

The **meta** display style outputs only the meta-data information associated with the result and discards the actual row results. This mode is useful for debugging the result sets generated from a full passthru Open Server gateway, or for those interested in what is really coming back from the server.

The **pretty** display style generates a fluffy table-like output using regular ASCII characters for borders. This mode does not perform any explicit column wrapping, like the

horiz display mode. However, the **$colwidth** variable can be used to control the maximum width of a given column on the screen. If the column exceeds **$colwidth** characters wide, it is wrapped in a relatively visually appealing manner. Note that **$colwidth** may be exceeded if there is enough screen width to hold the columns without wrapping.

The **none** display style suppresses all results from being displayed (however it does actually retrieve result information from the SQL Server). This is particularly useful when used with the **-p** flag (or the **$statistics** variable) for gathering accurate performance statistics.

thresh_display **(int)**
Sets the minimum SQL Server error severity that will display a message to the user, the default is 0 and valid ranges are between 0 and 22, inclusive.

thresh_exit **(int)** Defines the maximum number of errors of severity level **$thresh_fail** may be encountered before sqsh aborts. This is useful primarily for non-interactive scripts, but is allowed on an interactive session. Setting **$thresh_exit** to a value of 0 disables this feature. See section **EXIT STATUS** for details.

thresh_fail **(int)** Sets the minimum SQL Server severity level that is to be considered a failed batch. The minimum for this value is 0 (meaning any error that is not an information message), and the maximum is 22. Whenever **$thresh_fail** is crossed, the variable **$batch_failcount** is incremented by 1. See section **EXIT STATUS** for details.

time **(date-spec)** This variable may be set with a time format (see the man page for **date(1)**), and the variable expands to the current time in the supplied format. The default format for this variable is %H:%M:%S (e.g. 14:32:58).

tmp_dir **(path)** This contains the directory to which temporary files used internally by *sqsh* are to be written. These files are generated either during buffer editing (the **\buf−edit** command), or to maintain output defer files for background jobs. The default value for this variable is /tmp.

username **(string)**
The name of the user currently connected to the database.

version **(none)** This read-only variable contains the current version number.

width **(int)** The current width of the SQL output.

xgeom **(string/int)**
If X11 support is compiled into **sqsh**, this value is used to configure the default window size (in characters) of the X display. This variable must be of the format *WW*x*HH* or just *WW*, where *WW* is the width of the window and *HH* is the height of the window. If the height of the window is not supplied, then 25 lines is assumed. If **$xgeom** is not set, then **$width** is used as the default width and the height is assumed to be 25. If neither is set, then 80x25 is assumed.

SCRIPT EXECUTION

As with most shells, *sqsh* allows a file containing SQL and script commands to be executed directly via the magical UNIX **#!** convention.

On most UNIX platforms, when the operating system encounters the bytes **#!** as the first two bytes of an executable file it will automatically pipe the file through the interpreter specified immediately after the **#!**. For example, to create an executable *sqsh* script to run **sp_who**, you simply need to create a file like so:

```
#!/usr/local/bin/sqsh -i
sp_who
go
```

Thus, if your **sp_who** script is executed directly, it will automatically launch "**/usr/local/bin/sqsh -i sp_who**" for you.

And, to make things even more flexible, *sqsh* supports positional parameters, similar to most shells, of the form **${n}** which will expand to the **n**th argument to your *sqsh* script. For example:

```
#!/usr/local/bin/sqsh -i
sp_who ${1}
go
```

will cause the **sp_who** stored procedure to be executed with an argument of the first command line parameter supplied to the **sp_who** shell script.

Note that positional parameters *must* be contained between braces to avoid conflicts with the TSQL **money** data type (without the braces, the variable will not be expanded).

EXIT STATUS

One of the major complaints of *isql* is that it provides no facility to detect when an error condition occurred while it is performing processing. *sqsh* provides a rather complex, but flexible mechanism for returning meaningful information concerning its reason for exit in the form of an exit status (see **exit(3)**).

When *sqsh* begins execution two handlers are associated with the current connection to the database, one is a message handler which is responsible for displaying the text of any SQL Server messages or errors, and the other is an error handler, which is responsible for determining what to do with an error condition (bear with me, these are only loose descriptions). And, associated with each message and error condition is a severity level, between 0 and 22 (informational message to fatal condition).

Associated with these two message handlers are several variables that are used to either control their behavior, or are used as indicators by the message handler:

$thresh_display

This variable is used by the message handler to determine the minimum error severity which will cause a message to be displayed. By default this is 0, which will display all messages (with a couple of exceptions). Setting this to 1, for example, would suppress information messages such as the output of **set showplan**.

$thresh_fail

This variable is used by the error handler to determine which error severity is considered by *sqsh* to be a failure. Normally, this defaults to 11 which indicates that any error, other than informational messages, is a failure. The next variable will explain the importance of this value.

$batch_failcount

This variable should be considered read-only, and contains the total number of times that batches have caused an error of severity **$thresh_fail** or more. The only value that is valid to explicitly set this to is "" (the empty string), which will reset this value to 0, any other value may have unpredictable results.

$thresh_exit

This variable is used to determine the limit at which **$batch_failcount** will cause *sqsh* to exit. If **$thresh_exit** is 0, then this feature is disabled. In other words, if **$batch_failcount** == **$thresh_exit** and **$thresh_exit** is greater than 0, then *sqsh* will exit, returning **$batch_failcount** as an exit status.

Note that, unless **$exit_failcount** is set to 1, *sqsh* will exit with 0 if the total number of failures does not reach **$thresh_exit.**

$exit_failcount

This variable is used only when *sqsh* would normally exit with a success status (0), this causes it to instead exit with a value of **$batch_failout** (which may, itself, be 0).

To recap, here are a list of error codes that may be returned by *sqsh* upon exit, and the reason that they could be returned:

0	No error has been encountered.
1..253	Between 1 and 253 batches have failed (if you run more than 253 batches, the exit status of sqsh is undetermined...I may fix this in the future).
254	An explicit **\abort** was called, or a SIGINT (ˆC) was issued during a non-interactive session.
255	A general error condition has occurred, such as a bad command line argument to sqsh, memory allocation failure, file access error, etc.

The following sections provide detailed examples of combinations of variable settings and the results produced upon exit with certain failure conditions:

thresh_display=0, thresh_fail=0, thresh_exit=1
With this combination, all error messages will be displayed as they happen, and every error will be considered an failure condition. Upon reaching the first error, *sqsh* will abort with an exit status of 1, or the total number of failures (the **$batch_failcount** variable). However, if nothing goes wrong during the whole process, a zero is returned.

thresh_display=0, thresh_fail=0, thresh_exit=3
This combination will cause all error conditions to be displayed and all of them to be considered a failure condition. Upon reaching three total failed batches, *sqsh* with exit with a status of 3. However if 0, 1, or 2 batches fail, then 0 is returned.

thresh_display=22, thresh_fail=0, thresh_exit=3
This behaves the same as the previous example, with the exception that all error messages will be suppressed from being displayed. This is particularly useful if you just care about the exit value more than the actual error.

thresh_display=0, thresh_fail=2, thresh_exit=1
This will cause the first error of severity 2 or higher to be displayed and cause *sqsh* to exit with a failure condition of 1.

thresh_display=0, thresh_fail=0, thresh_exit=3, exit_failcount=1
This is identical to the second example, above, however *sqsh* will return the total number of batches that failed even if **$batch_failcount** does not reach 3.

FILES

$HOME/.sqshrc, **$HOME/.sqsh_session**, **$HOME/.sqsh_history**, **$HOME/.sqsh_readline**, **$HOME/.sqsh_words**, **$tmp_dir/sqsh-dfr.**∗, **$tmp_dir/sqsh-edit.**∗

BUGS

The addition of flow-of-control expressions has extended sqsh *way* beyond the scope of its original design, and it is quite obvious from using the features they are hacked in and are rather klunky (although still quite usable). As a result, the processing of these expressions is rather slow (when compared to bourne shell), and the error reporting doesn't lend itself to debugging large scripts. The development of 1000+ line scripts is discouraged.

The combination of backgrounding and pipes does not work properly right now. I know why this is happening, but haven't determined an elegant solution to it just yet. What happens is, when a background job is run that incorporates a pipe-line, *sqsh* will suspend until the job is complete, which is obviously not what you would desire. To test this, try the following:

```
1> select * from syscolumns
2> go | grep id &
```

You will find that you do not get your prompt back until the job completes. If you want a technical explanation of why this is happening, send me e-mail at the address at the end.

I would like to support all of the flags available in **isql** right now. This shouldn't be very hard.

No complaints about spelling or grammar. I hate documentation, so count yourself lucky that you have a manual page at all.

I know that there are more lurking out there; if you find any please send e-mail to gray@voicenet.com, or grays@xtend-tech.com and I'll jump on them.