NAME

expect − programmed dialogue with interactive programs, Version 5

SYNOPSIS

expect [**−dDinN**] [**−c** *cmds*] [[−[**f**|**b**]] *cmdfile*] [*args*]

INTRODUCTION

Expect is a program that "talks" to other interactive programs according to a script. Following the script, **Expect** knows what can be expected from a program and what the correct response should be. An interpreted language provides branching and high-level control structures to direct the dialogue. In addition, the user can take control and interact directly when desired, afterward returning control to the script.

Expectk is a mixture of **Expect** and **Tk**. It behaves just like **Expect** and **Tk**'s **wish**. **Expect** can also be used directly in C or C++ (that is, without Tcl). See libexpect(3).

The name "Expect" comes from the idea of *send/expect* sequences popularized by uucp, kermit and other modem control programs. However unlike uucp, **Expect** is generalized so that it can be run as a user-level command with any program and task in mind. **Expect** can actually talk to se veral programs at the same time.

For example, here are some things **Expect** can do:

- Cause your computer to dial you back, so that you can login without paying for the call.
- Start a game (e.g., rogue) and if the optimal configuration doesn't appear, restart it (again and again) until it does, then hand over control to you.
- Run fsck, and in response to its questions, answer "yes", "no" or give control back to you, based on predetermined criteria.
- Connect to another network or BBS (e.g., MCI Mail, CompuServe) and automatically retrieve your mail so that it appears as if it was originally sent to your local system.
- Carry environment variables, current directory, or any kind of information across rlogin, telnet, tip, su, chgrp, etc.

There are a variety of reasons why the shell cannot perform these tasks. (Try, you'll see.) All are possible with **Expect**.

In general, **Expect** is useful for running any program which requires interaction between the program and the user. All that is necessary is that the interaction can be characterized programmatically. **Expect** can also give the user back control (without halting the program being controlled) if desired. Similarly, the user can return control to the script at any time.

USAGE

Expect reads *cmdfile* for a list of commands to execute. **Expect** may also be invoked implicitly on systems which support the #! notation by marking the script executable, and making the first line in your script:

```
#!/usr/bin/expect −f
```

Of course, the path must accurately describe where **Expect** lives. /usr/bin is just an example.

The **−c** flag prefaces a command to be executed before any in the script. The command should be quoted to prevent being broken up by the shell. This option may be used multiple times. Multiple commands may be executed with a single **−c** by separating them with semicolons. Commands are executed in the order they appear. (When using Expectk, this option is specified as**−command**.)

The **−d** flag enables some diagnostic output, which primarily reports internal activity of commands such as **expect** and **interact**. This flag has the same effect as "exp_internal 1" at the beginning of an Expect script, plus the version of **Expect** is printed. (The **strace** command is useful for tracing statements, and the **trace** command is useful for tracing variable assignments.) (When using Expectk, this option is specified as **−diag**.)

The **−D** flag enables an interactive debugger. An integer value should follow. The debugger will take control before the next Tcl procedure if the value is non-zero or if a ^C is pressed (or a breakpoint is hit, or other appropriate debugger command appears in the script). See the README file or SEE ALSO (below) for more information on the debugger. (When using Expectk, this option is specified as**−Deb ug**.)

The **−f** flag prefaces a file from which to read commands from. The flag itself is optional as it is only useful when using the #! notation (see above), so that other arguments may be supplied on the command line. (When using Expectk, this option is specified as **−file**.)

By default, the command file is read into memory and executed in its entirety. It is occasionally desirable to read files one line at a time. For example, stdin is read this way. In order to force arbitrary files to be handled this way, use the **−b** flag. (When using Expectk, this option is specified as **−buffer**.) Note that stdio-buffering may still take place however this shouldn't cause problems when reading from a fifo or stdin.

If the string "−" is supplied as a filename, standard input is read instead. (Use "./−" to read from a file actually named "−".)

The **−i** flag causes **Expect** to interactively prompt for commands instead of reading them from a file. Prompting is terminated via the **exit** command or upon EOF. See**inter preter** (below) for more information. **−i** is assumed if neither a command file nor **−c** is used. (When using Expectk, this option is specified as **−interactive**.)

−− may be used to delimit the end of the options. This is useful if you want to pass an option-like argument to your script without it being interpreted by **Expect**. This can usefully be placed in the #! line to prevent any flag-like interpretation by Expect. For example, the following will leave the original arguments (including the script name) in the variable *argv*.

```
#!/usr/bin/expect −−
```

Note that the usual getopt(3) and execve(2) conventions must be observed when adding arguments to the #! line.

The file $exp_library/expect.rc is sourced automatically if present, unless the **−N** flag is used. (When using Expectk, this option is specified as **−NORC**.) Immediately after this, the file ~/.expect.rc is sourced automatically, unless the **−n** flag is used. If the environment variable DOTDIR is defined, it is treated as a directory and .expect.rc is read from there. (When using Expectk, this option is specified as **−norc**.) This sourcing occurs only after executing any **−c** flags.

−v causes Expect to print its version number and exit. (The corresponding flag in Expectk, which uses long flag names, is −version.)

Optional *args* are constructed into a list and stored in the variable named *argv*. *argc* is initialized to the length of argv.

argv0 is defined to be the name of the script (or binary if no script is used). For example, the following prints out the name of the script and the first three arguments:

```
send_user "$argv0 [lrange $argv 0 2]\n"
```

COMMANDS

Expect uses *Tcl* (Tool Command Language). Tcl provides control flow (e.g., if, for, break), expression evaluation and several other features such as recursion, procedure definition, etc. Commands used here but not defined (e.g., **set**, **if**, **exec**) are Tcl commands (see tcl(3)). **Expect** supports additional commands, described below. Unless otherwise specified, commands return the empty string.

Commands are listed alphabetically so that they can be quickly located. However, new users may find it easier to start by reading the descriptions of **spawn**, **send**, **expect**, and **interact**, in that order.

Note that the best introduction to the language (both Expect and Tcl) is provided in the book "Exploring

Expect" (see SEE ALSO below). Examples are included in this man page but they are very limited since this man page is meant primarily as reference material.

Note that in the text of this man page, "Expect" with an uppercase "E" refers to the **Expect** program while "expect" with a lower-case "e" refers to the **expect** command within the **Expect** program.)

close *[−slave] [−onexec 0|1] [−i spawn_id]*

closes the connection to the current process. Most interactive programs will detect EOF on their stdin and exit; thus **close** usually suffices to kill the process as well. The **−i** flag declares the process to close corresponding to the named spawn_id.

Both **expect** and **interact** will detect when the current process exits and implicitly do a **close**. But if you kill the process by, say, "exec kill $pid", you will need to explicitly call **close**.

The **−onexec** flag determines whether the spawn id will be closed in any new spawned processes or if the process is overlayed. To leave a spawn id open, use the value 0. A non-zero integer value will force the spawn closed (the default) in any new processes.

The **−slave** flag closes the slave associated with the spawn id. (See "spawn −pty".) When the connection is closed, the slave is automatically closed as well if still open.

No matter whether the connection is closed implicitly or explicitly, you should call **wait** to clear up the corresponding kernel process slot. **close** does not call **wait** since there is no guarantee that closing a process connection will cause it to exit. See**wait** belo w for more info.

debug *[[−now] 0|1]*

controls a Tcl debugger allowing you to step through statements, set breakpoints, etc.

With no arguments, a 1 is returned if the debugger is not running, otherwise a 0 is returned.

With a 1 argument, the debugger is started. With a 0 argument, the debugger is stopped. If a 1 argument is preceded by the **−now** flag, the debugger is started immediately (i.e., in the middle of the **debug** command itself). Otherwise, the debugger is started with the next Tcl statement.

The **debug** command does not change any traps. Compare this to starting Expect with the **−D** flag (see above).

See the README file or SEE ALSO (below) for more information on the debugger.

disconnect

disconnects a forked process from the terminal. It continues running in the background. The process is given its own process group (if possible). Standard I/O is redirected to /dev/null.

The following fragment uses **disconnect** to continue running the script in the background.

```
if {[fork]!=0} exit
disconnect
. . .
```

The following script reads a password, and then runs a program every hour that demands a password each time it is run. The script supplies the password so that you only have to type it once. (See the **stty** command which demonstrates how to turn off password echoing.)

```
send_user "password?\ "
expect_user −re "(.*)\n"
for {} 1 {} {
   if {[fork]!=0} {sleep 3600;continue}
```

```
    disconnect
    spawn priv_prog
    expect Password:
    send "$expect_out(1,string)\r"
    . . .
    exit
}
```

An advantage to using **disconnect** over the shell asynchronous process feature (&) is that **Expect** can save the terminal parameters prior to disconnection, and then later apply them to new ptys. With &, **Expect** does not have a chance to read the terminal's parameters since the terminal is already disconnected by the time **Expect** receives control.

exit *[−opts] [status]*

causes **Expect** to exit or otherwise prepare to do so.

The **−onexit** flag causes the next argument to be used as an exit handler. Without an argument, the current exit handler is returned.

The **−noexit** flag causes **Expect** to prepare to exit but stop short of actually returning control to the operating system. The user-defined exit handler is run as well as Expect's own internal handlers. No further Expect commands should be executed. This is useful if you are running Expect with other Tcl extensions. The current interpreter (and main window if in the Tk environment) remain so that other Tcl extensions can clean up. If Expect's **exit** is called again (however this might occur), the handlers are not rerun.

Upon exiting, all connections to spawned processes are closed. Closure will be detected as an EOF by spawned processes. **exit** tak es no other actions beyond what the normal _exit(2) procedure does. Thus, spawned processes that do not check for EOF may continue to run. (A variety of conditions are important to determining, for example, what signals a spawned process will be sent, but these are system-dependent, typically documented under exit(3).) Spawned processes that continue to run will be inherited by init.

status (or 0 if not specified) is returned as the exit status of **Expect**. **exit** is implicitly executed if the end of the script is reached.

exp_continue [−continue_timer]

The command **exp_continue** allows **expect** itself to continue executing rather than returning as it normally would. By default **exp_continue** resets the timeout timer. The *−continue_timer* flag prevents timer from being restarted. (See **expect** for more information.)

exp_internal *[−f file] value*

causes further commands to send diagnostic information internal to **Expect** to stderr if *value* is nonzero. This output is disabled if*value* is 0. The diagnostic information includes e very character received, and every attempt made to match the current output against the patterns.

If the optional *file* is supplied, all normal and deb ugging output is written to that file (regardless of the value of *value*). Any previous diagnostic output file is closed.

The **−info** flag causes exp_internal to return a description of the most recent non-info arguments given.

exp_open *[args] [−i spawn_id]*

returns a Tcl file identifier that corresponds to the original spawn id. The file identifier can then be used as if it were opened by Tcl's **open** command. (The spawn id should no longer be used. A **wait** should not be executed.

The **−leaveopen** flag leaves the spawn id open for access through Expect commands. A **wait** must be executed on the spawn id.

exp_pid *[−i spawn_id]*
returns the process id corresponding to the currently spawned process. If the **−i** flag is used, the pid returned corresponds to that of the given spawn id.

exp_send
is an alias for **send**.

exp_send_error
is an alias for **send_error**.

exp_send_log
is an alias for **send_log**.

exp_send_tty
is an alias for **send_tty**.

exp_send_user
is an alias for **send_user**.

exp_version *[[−exit] version]*
is useful for assuring that the script is compatible with the current version of Expect.

With no arguments, the current version of **Expect** is returned. This version may then be encoded in your script. If you actually know that you are not using features of recent versions, you can specify an earlier version.

Versions consist of three numbers separated by dots. First is the major number. Scripts written for versions of **Expect** with a different major number will almost certainly not work. **exp_version** returns an error if the major numbers do not match.

Second is the minor number. Scripts written for a version with a greater minor number than the current version may depend upon some new feature and might not run. **exp_v ersion** returns an error if the major numbers match, but the script minor number is greater than that of the running **Expect**.

Third is a number that plays no part in the version comparison. However, it is incremented when the **Expect** software distribution is changed in any way, such as by additional documentation or optimization. It is reset to 0 upon each new minor version.

With the **−exit** flag, **Expect** prints an error and exits if the version is out of date.

expect *[[−opts] pat1 body1] ... [−opts] patn [bodyn]*
waits until one of the patterns matches the output of a spawned process, a specified time period has passed, or an end-of-file is seen. If the final body is empty, it may be omitted.

Patterns from the most recent **expect_before** command are implicitly used before any other patterns. Patterns from the most recent **expect_after** command are implicitly used after any other patterns.

If the arguments to the entire **expect** statement require more than one line, all the arguments may be "braced" into one so as to avoid terminating each line with a backslash. In this one case, the usual Tcl substitutions will occur despite the braces.

If a pattern is the keyword **eof**, the corresponding body is executed upon end-of-file. If a pattern is the keyword **timeout**, the corresponding body is executed upon timeout. If no timeout keyword is used, an implicit null action is executed upon timeout. The default timeout period is 10 seconds but may be set, for example to 30, by the command "set timeout 30". An infinite timeout may be designated by the value −1. If a pattern is the keyword **default**, the corresponding body is executed upon either timeout or end-of-file.

If a pattern matches, then the corresponding body is executed. **expect** returns the result of the body (or the empty string if no pattern matched). In the event that multiple patterns match, the one appearing first is used to select a body.

Each time new output arrives, it is compared to each pattern in the order they are listed. Thus, you may test for absence of a match by making the last pattern something guaranteed to appear, such as a prompt. In situations where there is no prompt, you must use **timeout** (just like you would if you were interacting manually).

Patterns are specified in three ways. By default, patterns are specified as with Tcl's **string match** command. (Such patterns are also similar to C-shell regular expressions usually referred to as "glob" patterns). The **−gl** flag may may be used to protect patterns that might otherwise match **expect** flags from doing so. Any pattern beginning with a "−" should be protected this way. (All strings starting with "−" are reserved for future options.)

For example, the following fragment looks for a successful login. (Note that **abort** is presumed to be a procedure defined elsewhere in the script.)

```
expect {
   busy             {puts busy\n ; exp_continue}
   failed           abort
   "invalid password" abort
   timeout          abort
   connected
}
```

Quotes are necessary on the fourth pattern since it contains a space, which would otherwise separate the pattern from the action. Patterns with the same action (such as the 3rd and 4th) require listing the actions again. This can be avoid by using regexp-style patterns (see below). More information on forming glob-style patterns can be found in the Tcl manual.

Regexp-style patterns follow the syntax defined by Tcl's **regexp** (short for "regular expression") command. regexp patterns are introduced with the flag **−re**. The previous example can be rewritten using a regexp as:

```
expect {
   busy      {puts busy\n ; exp_continue}
   −re "failed|invalid password" abort
   timeout   abort
   connected
}
```

Both types of patterns are "unanchored". This means that patterns do not have to match the entire string, but can begin and end the match anywhere in the string (as long as everything else matches). Use ^ to match the beginning of a string, and $ to match the end. Note that if you do not wait for the end of a string, your responses can easily end up in the middle of the string as they are echoed from the spawned process. While still producing correct results, the output can look unnatural. Thus, use of $ is encouraged if you can exactly describe the characters at the end of a string.

Note that in many editors, the ^ and $ match the beginning and end of lines respectively. However, because expect is not line oriented, these characters match the beginning and end of the data (as opposed to lines) currently in the expect matching buffer. (Also, see the note below on "system indigestion.")

The **−ex** flag causes the pattern to be matched as an "exact" string. No interpretation of *, ^, etc is made (although the usual Tcl conventions must still be observed). Exact patterns are always unanchored.

The −**nocase** flag causes uppercase characters of the output to compare as if they were lowercase characters. The pattern is not affected.

While reading output, more than 2000 bytes can force earlier bytes to be "forgotten". This may be changed with the function **match_max**. (Note that excessively large values can slow down the pattern matcher.) If*patlist* is **full_b uffer**, the corresponding body is executed if *match_max* bytes have been received and no other patterns have matched. Whether or not the **full_buffer** keyword is used, the forgotten characters are written to expect_out(buffer).

If *patlist* is the keyword **null**, and nulls are allowed (via the **remove_nulls** command), the corresponding body is executed if a single ASCII 0 is matched. It is not possible to match 0 bytes via glob or regexp patterns.

Upon matching a pattern (or eof or full_buffer), any matching and previously unmatched output is saved in the variable *expect_out(buffer)*. Up to 9 regexp substring matches are saved in the variables *expect_out(1,string)* through *expect_out(9,string)*. If the−**indices** flag is used before a pattern, the starting and ending indices (in a form suitable for **lrange**) of the 10 strings are stored in the variables *expect_out(X,start)* and *expect_out(X,end)* where X is a digit, corresponds to the substring position in the buffer. 0 refers to strings which matched the entire pattern and is generated for glob patterns as well as regexp patterns. For example, if a process has produced output of "abcdefgh\n", the result of:

```
expect "cd"
```

is as if the following statements had executed:

```
set expect_out(0,string) cd
set expect_out(buffer) abcd
```

and "efgh\n" is left in the output buffer. If a process produced the output "abbbcabkkkka\n", the result of:

```
expect −indices −re "b(b*).*(k+)"
```

is as if the following statements had executed:

```
set expect_out(0,start) 1
set expect_out(0,end) 10
set expect_out(0,string) bbbcabkkkk
set expect_out(1,start) 2
set expect_out(1,end) 3
set expect_out(1,string) bb
set expect_out(2,start) 10
set expect_out(2,end) 10
set expect_out(2,string) k
set expect_out(buffer) abbbcabkkkk
```

and "a\n" is left in the output buffer. The pattern "*" (and −re ".*") will flush the output buffer without reading any more output from the process.

Normally, the matched output is discarded from Expect's internal buffers. This may be prevented by prefixing a pattern with the −**notransfer** flag. This flag is especially useful in experimenting (and can be abbreviated to "−not" for convenience while experimenting).

The spawn id associated with the matching output (or eof or full_buffer) is stored in *expect_out(spawn_id)*.

The **−timeout** flag causes the current expect command to use the following value as a timeout instead of using the value of the timeout variable.

By default, patterns are matched against output from the current process, however the **−i** flag declares the output from the named spawn_id list be matched against any following patterns (up to the next **−i**). The spawn_id list should either be a whitespace separated list of spawn_ids or a variable referring to such a list of spawn_ids.

For example, the following example waits for "connected" from the current process, or "busy", "failed" or "invalid password" from the spawn_id named by $proc2.

```
expect {
   −i $proc2 busy {puts busy\n ; exp_continue}
   −re "failed|invalid password" abort
   timeout abort
   connected
}
```

The value of the global variable *any_spawn_id* may be used to match patterns to any spawn_ids that are named with all other **−i** flags in the current **expect** command. The spawn_id from a **−i** flag with no associated pattern (i.e., followed immediately by another **−i**) is made available to any other patterns in the same **expect** command associated with *any_spawn_id*.

The **−i** flag may also name a global variable in which case the variable is read for a list of spawn ids. The variable is reread whenever it changes. This provides a way of changing the I/O source while the command is in execution. Spawn ids provided this way are called "indirect" spawn ids.

Actions such as **break** and **continue** cause control structures (i.e., **for**, **proc**) to behave in the usual way. The command**exp_continue** allo ws **expect** itself to continue executing rather than returning as it normally would.

This is useful for avoiding explicit loops or repeated expect statements. The following example is part of a fragment to automate rlogin. The **exp_continue** avoids having to write a second **expect** statement (to look for the prompt again) if the rlogin prompts for a password.

```
expect {
   Password: {
      stty −echo
      send_user "password (for $user) on $host: "
      expect_user −re "(.*)\n"
      send_user "\n"
      send "$expect_out(1,string)\r"
      stty echo
      exp_continue
   } incorrect {
      send_user "invalid password or account\n"
      exit
   } timeout {
      send_user "connection to $host timed out\n"
      exit
   } eof {
      send_user \
         "connection to host failed: $expect_out(buffer)"
      exit
   } −re $prompt
```

```
}
```

For example, the following fragment might help a user guide an interaction that is already totally automated. In this case, the terminal is put into raw mode. If the user presses "+", a variable is incremented. If "p" is pressed, several returns are sent to the process, perhaps to poke it in some way, and "i" lets the user interact with the process, effectively stealing away control from the script. In each case, the **exp_continue** allows the current **expect** to continue pattern matching after executing the current action.

```
stty raw −echo
expect_after {
    −i $user_spawn_id
    "p" {send "\r\r\r"; exp_continue}
    "+" {incr foo; exp_continue}
    "i" {interact; exp_continue}
    "quit" exit
}
```

By default, **exp_continue** resets the timeout timer. The timer is not restarted, if**exp_continue** is called with the **−continue_timer** flag.

expect_after *[expect_args]*

works identically to the **expect_before** except that if patterns from both **expect** and **expect_after** can match, the **expect** pattern is used. See the **expect_before** command for more information.

expect_background *[expect_args]*

takes the same arguments as **expect**, however it returns immediately. Patterns are tested whenever new input arrives. The pattern**timeout** and **default** are meaningless to **expect_backgr ound** and are silently discarded. Otherwise, the **expect_background** command uses **expect_before** and **expect_after** patterns just like **expect** does.

When **expect_background** actions are being evaluated, background processing for the same spawn id is blocked. Background processing is unblocked when the action completes. While background processing is blocked, it is possible to do a (foreground) **expect** on the same spawn id.

It is not possible to execute an **expect** while an **expect_background** is unblocked. **expect_background** for a particular spawn id is deleted by declaring a new expect_background with the same spawn id. Declaring **expect_background** with no pattern removes the given spawn id from the ability to match patterns in the background.

expect_before *[expect_args]*

takes the same arguments as **expect**, however it returns immediately. Pattern-action pairs from the most recent **expect_before** with the same spawn id are implicitly added to any following **expect** commands. If a pattern matches, it is treated as if it had been specified in the**expect** command itself, and the associated body is executed in the context of the **expect** command. If patterns from both **expect_before** and **expect** can match, the **expect_before** pattern is used.

If no pattern is specified, the spawn id is not checked for any patterns.

Unless overridden by a **−i** flag, **expect_before** patterns match against the spawn id defined at the time that the **expect_before** command was executed (not when its pattern is matched).

The −info flag causes **expect_before** to return the current specifications of what patterns it will match. By default, it reports on the current spawn id. An optional spawn id specification may be given for information on that spawn id. For example

```
expect_before −info −i $proc
```

At most one spawn id specification may be given. The flag −indirect suppresses direct spawn ids that come only from indirect specifications.

Instead of a spawn id specification, the flag "−all" will cause "−info" to report on all spawn ids.

The output of the −info flag can be reused as the argument to expect_before.

expect_tty *[expect_args]*
is like **expect** but it reads characters from /dev/tty (i.e. keystrokes from the user). By default, reading is performed in cooked mode. Thus, lines must end with a return in order for **expect** to see them. This may be changed via**stty** (see the **stty** command belo w).

expect_user *[expect_args]*
is like **expect** but it reads characters from stdin (i.e. keystrokes from the user). By default, reading is performed in cooked mode. Thus, lines must end with a return in order for **expect** to see them. This may be changed via **stty** (see the **stty** command below).

fork creates a new process. The new process is an exact copy of the current **Expect** process. On success, **fork** returns 0 to the new (child) process and returns the process ID of the child process to the parent process. On failure (invariably due to lack of resources, e.g., swap space, memory), **fork** returns −1 to the parent process, and no child process is created.

Forked processes exit via the **exit** command, just like the original process. Forked processes are allowed to write to the log files. If you do not disable debugging or logging in most of the processes, the result can be confusing.

Some pty implementations may be confused by multiple readers and writers, even momentarily. Thus, it is safest to **fork** before spawning processes.

interact *[string1 body1] ... [stringn [bodyn]]*
gives control of the current process to the user, so that keystrokes are sent to the current process, and the stdout and stderr of the current process are returned.

String-body pairs may be specified as arguments, in which case the body is executed when the corresponding string is entered. (By default, the string is not sent to the current process.) The **interpreter** command is assumed, if the final body is missing.

If the arguments to the entire **interact** statement require more than one line, all the arguments may be "braced" into one so as to avoid terminating each line with a backslash. In this one case, the usual Tcl substitutions will occur despite the braces.

For example, the following command runs interact with the following string-body pairs defined: When ^Z is pressed, **Expect** is suspended. (The **−reset** flag restores the terminal modes.) When ^A is pressed, the user sees "you typed a control-A" and the process is sent a ^A. When $ is pressed, the user sees the date. When ^C is pressed, **Expect** exits. If "foo" is entered, the user sees "bar". When ~~ is pressed, the **Expect** interpreter runs interactively.

```
set CTRLZ \032
interact {
   −reset $CTRLZ {exec kill −STOP [pid]}
   \001   {send_user "you typed a control−A\n";
           send "\001"
          }
   $      {send_user "The date is [clock format [clock seconds]]."}
   \003   exit
   foo    {send_user "bar"}
   ~~
}
```

In string-body pairs, strings are matched in the order they are listed as arguments. Strings that partially match are not sent to the current process in anticipation of the remainder coming. If characters are then entered such that there can no longer possibly be a match, only the part of the string will be sent to the process that cannot possibly begin another match. Thus, strings that are substrings of partial matches can match later, if the original strings that was attempting to be match ultimately fails.

By default, string matching is exact with no wild cards. (In contrast, the **expect** command uses glob-style patterns by default.) The **−ex** flag may be used to protect patterns that might otherwise match **interact** flags from doing so. Any pattern beginning with a "−" should be protected this way. (All strings starting with "−" are reserved for future options.)

The **−re** flag forces the string to be interpreted as a regexp-style pattern. In this case, matching substrings are stored in the variable *interact_out* similarly to the way **expect** stores its output in the variable **expect_out**. The **−indices** flag is similarly supported.

The pattern **eof** introduces an action that is executed upon end-of-file. A separate **eof** pattern may also follow the **−output** flag in which case it is matched if an eof is detected while writing output. The default **eof** action is "return", so that **interact** simply returns upon any EOF.

The pattern **timeout** introduces a timeout (in seconds) and action that is executed after no characters have been read for a given time. The **timeout** pattern applies to the most recently specified process. There is no default timeout. The special variable "timeout" (used by the **expect** command) has no affect on this timeout.

For example, the following statement could be used to autologout users who have not typed anything for an hour but who still get frequent system messages:

```
interact −input $user_spawn_id timeout 3600 return −output \
  $spawn_id
```

If the pattern is the keyword **null**, and nulls are allowed (via the **remove_nulls** command), the corresponding body is executed if a single ASCII 0 is matched. It is not possible to match 0 bytes via glob or regexp patterns.

Prefacing a pattern with the flag **−iwrite** causes the variable *interact_out(spawn_id)* to be set to the spawn_id which matched the pattern (or eof).

Actions such as **break** and **continue** cause control structures (i.e., **for**, **proc**) to behave in the usual way. However **return** causes interact to return to its caller, while **inter_return** causes **interact** to cause a return in its caller. For example, if "proc foo" called **interact** which then executed the action **inter_return**, **proc foo** would return. (This means that if **interact** calls **interpreter** interactively typing **return** will cause the interact to continue, while **inter_return** will cause the interact to return to its caller.)

During **interact**, raw mode is used so that all characters may be passed to the current process. If the current process does not catch job control signals, it will stop if sent a stop signal (by default ^Z). To restart it, send a continue signal (such as by "kill −CONT <pid>"). If you really want to send a SIGSTOP to such a process (by ^Z), consider spawning csh first and then running your program. On the other hand, if you want to send a SIGSTOP to **Expect** itself, first call interpreter (perhaps by using an escape character), and then press ^Z.

String-body pairs can be used as a shorthand for avoiding having to enter the interpreter and execute commands interactively. The previous terminal mode is used while the body of a string-body pair is being executed.

For speed, actions execute in raw mode by default. The **−r eset** flag resets the terminal to the mode it had before **interact** was executed (invariably, cooked mode). Note that characters entered when the mode is being switched may be lost (an unfortunate feature of the terminal driver on some systems). The only reason to use **−r eset** is if your action depends on running in cooked mode.

The **−echo** flag sends characters that match the following pattern back to the process that generated them as each character is read. This may be useful when the user needs to see feedback from partially typed patterns.

If a pattern is being echoed but eventually fails to match, the characters are sent to the spawned process. If the spawned process then echoes them, the user will see the characters twice. **−echo** is probably only appropriate in situations where the user is unlikely to not complete the pattern. For example, the following excerpt is from rftp, the recursive-ftp script, where the user is prompted to enter ~g, ~p, or ~l, to get, put, or list the current directory recursively. These are so far away from the normal ftp commands, that the user is unlikely to type ~ followed by anything else, except mistakenly, in which case, they'll probably just ignore the result anyway.

```
interact {
    −echo ~g {getcurdirectory 1}
    −echo ~l {getcurdirectory 0}
    −echo ~p {putcurdirectory}
}
```

The **−nobuffer** flag sends characters that match the following pattern on to the output process as characters are read.

This is useful when you wish to let a program echo back the pattern. For example, the following might be used to monitor where a person is dialing (a Hayes-style modem). Each time "atd" is seen the script logs the rest of the line.

```
proc lognumber {} {
    interact −nobuffer −re "(.*)\r" return
    puts $log "[clock format [clock seconds]]: dialed $interact_out(1,string)"
}

interact −nobuffer "atd" lognumber
```

During **interact**, previous use of **log_user** is ignored. In particular, **interact** will force its output to be logged (sent to the standard output) since it is presumed the user doesn't wish to interact blindly.

The **−o** flag causes any following key-body pairs to be applied to the output of the current process. This can be useful, for example, when dealing with hosts that send unwanted characters during a telnet session.

By default, **interact** expects the user to be writing stdin and reading stdout of the **Expect** process itself. The **−u** flag (for "user") mak es **interact** look for the user as the process named by its argument (which must be a spawned id).

This allows two unrelated processes to be joined together without using an explicit loop. To aid in debugging, Expect diagnostics always go to stderr (or stdout for certain logging and debugging information). For the same reason, the **interpreter** command will read interactively from stdin.

For example, the following fragment creates a login process. Then it dials the user (not shown), and finally connects the two together. Of course, any process may be substituted for login. A shell, for example, would allow the user to work without supplying an account and password.

```
spawn login
set login $spawn_id
```

```
spawn tip modem
# dial back out to user
# connect user to login
interact −u $login
```

To send output to multiple processes, list each spawn id list prefaced by a **−output** flag. Input for a group of output spawn ids may be determined by a spawn id list prefaced by a **−input** flag. (Both **−input** and **−output** may take lists in the same form as the **−i** flag in the **expect** command, except that any_spawn_id is not meaningful in **interact**.) All following flags and strings (or patterns) apply to this input until another −input flag appears. If no **−input** appears, **−output** implies "−input $user_spawn_id −output". (Similarly, with patterns that do not have **−input**.) If one**−input** is specified, it overrides $user_spawn_id. If a second**−input** is specified, it o verrides $spawn_id. Additional **−input** flags may be specified.

The two implied input processes default to having their outputs specified as $spawn_id and $user_spawn_id (in reverse). If a**−input** flag appears with no **−output** flag, characters from that process are discarded.

The **−i** flag introduces a replacement for the current spawn_id when no other **−input** or **−output** flags are used. A −i flag implies a −o flag.

It is possible to change the processes that are being interacted with by using indirect spawn ids. (Indirect spawn ids are described in the section on the expect command.) Indirect spawn ids may be specified with the −i, −u, −input, or −output flags.

interpreter [args]

causes the user to be interactively prompted for **Expect** and Tcl commands. The result of each command is printed.

Actions such as **break** and **continue** cause control structures (i.e., **for**, **proc**) to behave in the usual way. However **return** causes interpreter to return to its caller, while **inter_return** causes **interpreter** to cause a return in its caller. For example, if "proc foo" called **interpreter** which then executed the action **inter_return**, **proc foo** would return. Any other command causes **interpreter** to continue prompting for new commands.

By default, the prompt contains two integers. The first integer describes the depth of the evaluation stack (i.e., how many times Tcl_Eval has been called). The second integer is the Tcl history identifier. The prompt can be set by defining a procedure called "prompt1" whose return value becomes the next prompt. If a statement has open quotes, parens, braces, or brackets, a secondary prompt (by default "+> ") is issued upon newline. The secondary prompt may be set by defining a procedure called "prompt2".

During **interpreter**, cooked mode is used, even if the its caller was using raw mode.

If stdin is closed, **interpreter** will return unless the **−eof** flag is used, in which case the subsequent argument is invoked.

log_file *[args] [[−a] file]*

If a filename is provided, **log_file** will record a transcript of the session (beginning at that point) in the file. **log_file** will stop recording if no argument is given. Any previous log file is closed.

Instead of a filename, a Tcl file identifier may be provided by using the **−open** or **−leaveopen** flags. This is similar to the **spawn** command. (See **spawn** for more info.)

The **−a** flag forces output to be logged that was suppressed by the **log_user** command.

By default, the **log_file** command *appends* to old files rather than truncating them, for the convenience of being able to turn logging off and on multiple times in one session. To truncate files, use

the **−noappend** flag.

The **−info** flag causes log_file to return a description of the most recent non-info arguments given.

log_user *−info|0|1*

By default, the send/expect dialogue is logged to stdout (and a logfile if open). The logging to stdout is disabled by the command "log_user 0" and reenabled by "log_user 1". Logging to the logfile is unchanged.

The **−info** flag causes log_user to return a description of the most recent non-info arguments given.

match_max *[−d] [−i spawn_id] [size]*

defines the size of the buffer (in bytes) used internally by **expect**. With no *size* argument, the current size is returned.

With the **−d** flag, the default size is set. (The initial default is 2000.) With the **−i** flag, the size is set for the named spawn id, otherwise it is set for the current process.

overlay *[−# spawn_id] [−# spawn_id] [...] program [args]*

executes *program args* in place of the current **Expect** program, which terminates. A bare hyphen argument forces a hyphen in front of the command name as if it was a login shell. All spawn_ids are closed except for those named as arguments. These are mapped onto the named file identifiers.

Spawn_ids are mapped to file identifiers for the new program to inherit. For example, the following line runs chess and allows it to be controlled by the current process − say, a chess master.

```
overlay −0 $spawn_id −1 $spawn_id −2 $spawn_id chess
```

This is more efficient than "interact −u", however, it sacrifices the ability to do programmed interaction since the **Expect** process is no longer in control.

Note that no controlling terminal is provided. Thus, if you disconnect or remap standard input, programs that do job control (shells, login, etc) will not function properly.

parity *[−d] [−i spawn_id] [value]*

defines whether parity should be retained or stripped from the output of spawned processes. If *value* is zero, parity is stripped, otherwise it is not stripped. With no *value* argument, the current value is returned.

With the **−d** flag, the default parity value is set. (The initial default is 1, i.e., parity is not stripped.) With the **−i** flag, the parity value is set for the named spawn id, otherwise it is set for the current process.

remove_nulls *[−d] [−i spawn_id] [value]*

defines whether nulls are retained or removed from the output of spawned processes before pattern matching or storing in the variable *expect_out* or *interact_out*. If*value* is 1, nulls are remo ved. If *value* is 0, nulls are not removed. With no *value* argument, the current value is returned.

With the **−d** flag, the default value is set. (The initial default is 1, i.e., nulls are removed.) With the **−i** flag, the value is set for the named spawn id, otherwise it is set for the current process.

Whether or not nulls are removed, **Expect** will record null bytes to the log and stdout.

send *[−flags] string*

Sends *string* to the current process. For example, the command

```
send "hello world\r"
```

sends the characters, h e l l o <blank> w o r l d <return> to the current process. (Tcl includes a printf-like command (called **format**) which can build arbitrarily complex strings.)

Characters are sent immediately although programs with line-buffered input will not read the characters until a return character is sent. A return character is denoted "\r".

The **−−** flag forces the next argument to be interpreted as a string rather than a flag. Any string can be preceded by "−−" whether or not it actually looks like a flag. This provides a reliable mechanism to specify variable strings without being tripped up by those that accidentally look like flags. (All strings starting with "−" are reserved for future options.)

The **−i** flag declares that the string be sent to the named spawn_id. If the spawn_id is *user_spawn_id*, and the terminal is in raw mode, newlines in the string are translated to return-newline sequences so that they appear as if the terminal was in cooked mode. The **−raw** flag disables this translation.

The **−null** flag sends null characters (0 bytes). By default, one null is sent. An integer may follow the **−null** to indicate how many nulls to send.

The **−break** flag generates a break condition. This only makes sense if the spawn id refers to a tty device opened via "spawn −open". If you have spawned a process such as tip, you should use tip's convention for generating a break.

The **−s** flag forces output to be sent "slowly", thus avoid the common situation where a computer outtypes an input buffer that was designed for a human who would never outtype the same buffer. This output is controlled by the value of the variable "send_slow" which takes a two element list. The first element is an integer that describes the number of bytes to send atomically. The second element is a real number that describes the number of seconds by which the atomic sends must be separated. For example, "set send_slow {10 .001}" would force "send −s" to send strings with 1 millisecond in between each 10 characters sent.

The **−h** flag forces output to be sent (somewhat) like a human actually typing. Human-like delays appear between the characters. (The algorithm is based upon a Weibull distribution, with modifications to suit this particular application.) This output is controlled by the value of the variable "send_human" which takes a five element list. The first two elements are average interarrival time of characters in seconds. The first is used by default. The second is used at word endings, to simulate the subtle pauses that occasionally occur at such transitions. The third parameter is a measure of variability where .1 is quite variable, 1 is reasonably variable, and 10 is quite invariable. The extremes are 0 to infinity. The last two parameters are, respectively, a minimum and maximum interarrival time. The minimum and maximum are used last and "clip" the final time. The ultimate average can be quite different from the given average if the minimum and maximum clip enough values.

As an example, the following command emulates a fast and consistent typist:

```
set send_human {.1 .3 1 .05 2}
send −h "I'm hungry.  Let's do lunch."
```

while the following might be more suitable after a hangover:

```
set send_human {.4 .4 .2 .5 100}
send −h "Goodd party lash night!"
```

Note that errors are not simulated, although you can set up error correction situations yourself by embedding mistakes and corrections in a send argument.

The flags for sending null characters, for sending breaks, for forcing slow output and for human-style output are mutually exclusive. Only the one specified last will be used. Furthermore, no *string*

argument can be specified with the flags for sending null characters or breaks.

It is a good idea to precede the first **send** to a process by an **expect**. **expect** will wait for the process to start, while **send** cannot. In particular, if the first **send** completes before the process starts running, you run the risk of having your data ignored. In situations where interactive programs offer no initial prompt, you can precede **send** by a delay as in:

```
# To avoid giving hackers hints on how to break in,
# this system does not prompt for an external password.
# Wait for 5 seconds for exec to complete
spawn telnet very.secure.gov
sleep 5
send password\r
```

exp_send is an alias for **send**. If you are using Expectk or some other variant of Expect in the Tk environment, **send** is defined by Tk for an entirely different purpose. **exp_send** is pro vided for compatibility between environments. Similar aliases are provided for other Expect's other send commands.

send_error *[−flags] string*
is like **send**, except that the output is sent to stderr rather than the current process.

send_log *[−−] string*
is like **send**, except that the string is only sent to the log file (see **log_file**.) The arguments are ignored if no log file is open.

send_tty *[−flags] string*
is like **send**, except that the output is sent to /dev/tty rather than the current process.

send_user *[−flags] string*
is like **send**, except that the output is sent to stdout rather than the current process.

sleep *seconds*
causes the script to sleep for the given number of seconds. Seconds may be a decimal number. Interrupts (and Tk events if you are using Expectk) are processed while Expect sleeps.

spawn *[args] program [args]*
creates a new process running *program args*. Its stdin, stdout and stderr are connected to Expect, so that they may be read and written by other **Expect** commands. The connection is broken by **close** or if the process itself closes any of the file identifiers.

When a process is started by **spawn**, the variable *spawn_id* is set to a descriptor referring to that process. The process described by*spawn_id* is considered the *curr ent process*. *spawn_id* may be read or written, in effect providing job control.

user_spawn_id is a global variable containing a descriptor which refers to the user. For example, when *spawn_id* is set to this value, **expect** behaves like **expect_user**.

error_spawn_id is a global variable containing a descriptor which refers to the standard error. For example, when *spawn_id* is set to this value, **send** behaves like **send_error**.

tty_spawn_id is a global variable containing a descriptor which refers to /dev/tty. If /dev/tty does not exist (such as in a cron, at, or batch script), then *tty_spawn_id* is not defined. This may be tested as:

```
if {[info vars tty_spawn_id]} {
   # /dev/tty exists
} else {
   # /dev/tty doesn't exist
   # probably in cron, batch, or at script
```

```
}
```

spawn returns the UNIX process id. If no process is spawned, 0 is returned. The variable *spawn_out(slave,name)* is set to the name of the pty slave device.

By default, **spawn** echoes the command name and arguments. The **−noecho** flag stops **spawn** from doing this.

The **−console** flag causes console output to be redirected to the spawned process. This is not supported on all systems.

Internally, **spawn** uses a pty, initialized the same way as the user's tty. This is further initialized so that all settings are "sane" (according to stty(1)). If the variable *stty_init* is defined, it is interpreted in the style of stty arguments as further configuration. For example, "set stty_init raw" will cause further spawned processes's terminals to start in raw mode. **−nottycopy** skips the initialization based on the user's tty. **−nottyinit** skips the "sane" initialization.

Normally, **spawn** takes little time to execute. If you notice spawn taking a significant amount of time, it is probably encountering ptys that are wedged. A number of tests are run on ptys to avoid entanglements with errant processes. (These take 10 seconds per wedged pty.) Running Expect with the **−d** option will show if **Expect** is encountering many ptys in odd states. If you cannot kill the processes to which these ptys are attached, your only recourse may be to reboot.

If *program* cannot be spawned successfully because exec(2) fails (e.g. when *program* doesn't exist), an error message will be returned by the next **interact** or **expect** command as if *program* had run and produced the error message as output. This behavior is a natural consequence of the implementation of **spawn**. Internally, spawn forks, after which the spawned process has no way to communicate with the original **Expect** process except by communication via the spawn_id.

The **−open** flag causes the next argument to be interpreted as a Tcl file identifier (i.e., returned by **open**.) The spawn id can then be used as if it were a spawned process. (The file identifier should no longer be used.) This lets you treat raw devices, files, and pipelines as spawned processes without using a pty. 0 is returned to indicate there is no associated process. When the connection to the spawned process is closed, so is the Tcl file identifier. The **−lea veopen** flag is similar to **−open** except that **−leaveopen** causes the file identifier to be left open even after the spawn id is closed.

The **−pty** flag causes a pty to be opened but no process spawned. 0 is returned to indicate there is no associated process. Spawn_id is set as usual.

The variable *spawn_out(slave,fd)* is set to a file identifier corresponding to the pty slave. It can be closed using "close −slave".

The **−ignore** flag names a signal to be ignored in the spawned process. Otherwise, signals get the default behavior. Signals are named as in the **trap** command, e xcept that each signal requires a separate flag.

strace *level*

causes following statements to be printed before being executed. (Tcl's trace command traces variables.) *level* indicates how far down in the call stack to trace. For example, the following command runs **Expect** while tracing the first 4 levels of calls, but none below that.

```
expect −c "strace 4" script.exp
```

The **−info** flag causes strace to return a description of the most recent non-info arguments given.

stty *args*

changes terminal modes similarly to the external stty command.

By default, the controlling terminal is accessed. Other terminals can be accessed by appending "< /dev/tty..." to the command. (Note that the arguments should not be grouped into a single argument.)

Requests for status return it as the result of the command. If no status is requested and the controlling terminal is accessed, the previous status of the raw and echo attributes are returned in a form which can later be used by the command.

For example, the arguments **raw** or **−cooked** put the terminal into raw mode. The arguments **−raw** or **cooked** put the terminal into cooked mode. The arguments **echo** and **−echo** put the terminal into echo and noecho mode respectively.

The following example illustrates how to temporarily disable echoing. This could be used in otherwise-automatic scripts to avoid embedding passwords in them. (See more discussion on this under EXPECT HINTS below.)

```
stty −echo
send_user "Password: "
expect_user −re "(.*)\n"
set password $expect_out(1,string)
stty echo
```

system *args*

gives *args* to sh(1) as input, just as if it had been typed as a command from a terminal. **Expect** waits until the shell terminates. The return status from sh is handled the same way that **exec** handles its return status.

In contrast to **exec** which redirects stdin and stdout to the script, **system** performs no redirection (other than that indicated by the string itself). Thus, it is possible to use programs which must talk directly to /dev/tty. For the same reason, the results of **system** are not recorded in the log.

timestamp *[args]*

returns a timestamp. With no arguments, the number of seconds since the epoch is returned.

The **−format** flag introduces a string which is returned but with substitutions made according to the POSIX rules for strftime. For example %a is replaced by an abbreviated weekday name (i.e., Sat). Others are:

```
%a    abbreviated weekday name
%A    full weekday name
%b    abbreviated month name
%B    full month name
%c    date-time as in: Wed Oct  6 11:45:56 1993
%d    day of the month (01-31)
%H    hour (00-23)
%I    hour (01-12)
%j    day (001-366)
%m    month (01-12)
%M    minute (00-59)
%p    am or pm
%S    second (00-61)
%u    day (1-7, Monday is first day of week)
%U    week (00-53, first Sunday is first day of week one)
%V    week (01-53, ISO 8601 style)
```

%w day (0-6)
%W week (00-53, first Monday is first day of week one)
%x date-time as in: Wed Oct 6 1993
%X time as in: 23:59:59
%y year (00-99)
%Y year as in: 1993
%Z timezone (or nothing if not determinable)
%% a bare percent sign

Other % specifications are undefined. Other characters will be passed through untouched. Only the C locale is supported.

The **−seconds** flag introduces a number of seconds since the epoch to be used as a source from which to format. Otherwise, the current time is used.

The **−gmt** flag forces timestamp output to use the GMT timezone. With no flag, the local timezone is used.

trap *[[command] signals]*

causes the given *command* to be executed upon future receipt of any of the given signals. The command is executed in the global scope. If *command* is absent, the signal action is returned. If *command* is the string SIG_IGN, the signals are ignored. If *command* is the string SIG_DFL, the signals are result to the system default. *signals* is either a single signal or a list of signals. Signals may be specified numerically or symbolically as per signal(3). The "SIG" prefix may be omitted.

With no arguments (or the argument −number), **trap** returns the signal number of the trap command currently being executed.

The **−code** flag uses the return code of the command in place of whatever code Tcl was about to return when the command originally started running.

The **−interp** flag causes the command to be evaluated using the interpreter active at the time the command started running rather than when the trap was declared.

The **−name** flag causes the **trap** command to return the signal name of the trap command currently being executed.

The **−max** flag causes the **trap** command to return the largest signal number that can be set.

For example, the command "trap {send_user "Ouch!"} SIGINT" will print "Ouch!" each time the user presses ^C.

By default, SIGINT (which can usually be generated by pressing ^C) and SIGTERM cause Expect to exit. This is due to the following trap, created by default when Expect starts.

```
trap exit {SIGINT SIGTERM}
```

If you use the −D flag to start the debugger, SIGINT is redefined to start the interactive debugger. This is due to the following trap:

```
trap {exp_debug 1} SIGINT
```

The debugger trap can be changed by setting the environment variable EXPECT_DEBUG_INIT to a new trap command.

You can, of course, override both of these just by adding trap commands to your script. In particular, if you have your own "trap exit SIGINT", this will override the debugger trap. This is useful if you want to prevent users from getting to the debugger at all.

If you want to define your own trap on SIGINT but still trap to the debugger when it is running, use:

```
if {![exp_debug]} {trap mystuff SIGINT}
```

Alternatively, you can trap to the debugger using some other signal.

trap will not let you override the action for SIGALRM as this is used internally to **Expect**. The disconnect command sets SIGALRM to SIG_IGN (ignore). You can reenable this as long as you disable it during subsequent spawn commands.

See signal(3) for more info.

wait *[args]*

delays until a spawned process (or the current process if none is named) terminates.

wait normally returns a list of four integers. The first integer is the pid of the process that was waited upon. The second integer is the corresponding spawn id. The third integer is −1 if an operating system error occurred, or 0 otherwise. If the third integer was 0, the fourth integer is the status returned by the spawned process. If the third integer was −1, the fourth integer is the value of errno set by the operating system. The global variable errorCode is also set.

Additional elements may appear at the end of the return value from **wait**. An optional fifth element identifies a class of information. Currently, the only possible value for this element is CHILDKILLED in which case the next two values are the C-style signal name and a short textual description.

The **−i** flag declares the process to wait corresponding to the named spawn_id (NOT the process id). Inside a SIGCHLD handler, it is possible to wait for any spawned process by using the spawn id −1.

The **−nowait** flag causes the wait to return immediately with the indication of a successful wait. When the process exits (later), it will automatically disappear without the need for an explicit wait.

The **wait** command may also be used wait for a forked process using the arguments "−i −1". Unlike its use with spawned processes, this command can be executed at any time. There is no control over which process is reaped. However, the return value can be checked for the process id.

LIBRARIES

Expect automatically knows about two built-in libraries for Expect scripts. These are defined by the directories named in the variables exp_library and exp_exec_library. Both are meant to contain utility files that can be used by other scripts.

exp_library contains architecture-independent files. exp_exec_library contains architecture-dependent files. Depending on your system, both directories may be totally empty. The existence of the file $exp_exec_library/cat-buffers describes whether your /bin/cat buffers by default.

PRETTY-PRINTING

A vgrind definition is available for pretty-printing **Expect** scripts. Assuming the vgrind definition supplied with the **Expect** distribution is correctly installed, you can use it as:

```
vgrind −lexpect file
```

EXAMPLES

It many not be apparent how to put everything together that the man page describes. I encourage you to read and try out the examples in the example directory of the **Expect** distribution. Some of them are real programs. Others are simply illustrative of certain techniques, and of course, a couple are just quick hacks. The INSTALL file has a quick overview of these programs.

The **Expect** papers (see SEE ALSO) are also useful. While some papers use syntax corresponding to earlier versions of Expect, the accompanying rationales are still valid and go into a lot more detail than this man page.

CAVEATS

Extensions may collide with Expect's command names. For example, **send** is defined by Tk for an entirely different purpose. For this reason, most of the **Expect** commands are also available as "exp_XXXX". Commands and variables beginning with "exp", "inter", "spawn", and "timeout" do not have aliases. Use the extended command names if you need this compatibility between environments.

Expect takes a rather liberal view of scoping. In particular, variables read by commands specific to the **Expect** program will be sought first from the local scope, and if not found, in the global scope. For example, this obviates the need to place "global timeout" in every procedure you write that uses **expect**. On the other hand, variables written are always in the local scope (unless a "global" command has been issued). The most common problem this causes is when spawn is executed in a procedure. Outside the procedure, *spawn_id* no longer exists, so the spawned process is no longer accessible simply because of scoping. Add a "global spawn_id" to such a procedure.

If you cannot enable the multispawning capability (i.e., your system supports neither select (BSD *.*), poll (SVR>2), nor something equivalent), **Expect** will only be able to control a single process at a time. In this case, do not attempt to set *spawn_id*, nor should you execute processes via exec while a spawned process is running. Furthermore, you will not be able to**expect** from multiple processes (including the user as one) at the same time.

Terminal parameters can have a big effect on scripts. For example, if a script is written to look for echoing, it will misbehave if echoing is turned off. For this reason, Expect forces sane terminal parameters by default. Unfortunately, this can make things unpleasant for other programs. As an example, the emacs shell wants to change the "usual" mappings: newlines get mapped to newlines instead of carriage-return newlines, and echoing is disabled. This allows one to use emacs to edit the input line. Unfortunately, Expect cannot possibly guess this.

You can request that Expect not override its default setting of terminal parameters, but you must then be very careful when writing scripts for such environments. In the case of emacs, avoid depending upon things like echoing and end-of-line mappings.

The commands that accepted arguments braced into a single list (the **expect** variants and **interact**) use a heuristic to decide if the list is actually one argument or many. The heuristic can fail only in the case when the list actually does represent a single argument which has multiple embedded \n's with non-whitespace characters between them. This seems sufficiently improbable, however the argument "−nobrace" can be used to force a single argument to be handled as a single argument. This could conceivably be used with machine-generated Expect code. Similarly, -brace forces a single argument to be handle as multiple patterns/actions.

BUGS

It was really tempting to name the program "sex" (for either "Smart EXec" or "Send-EXpect"), but good sense (or perhaps just Puritanism) prevailed.

On some systems, when a shell is spawned, it complains about not being able to access the tty but runs anyway. This means your system has a mechanism for gaining the controlling tty that **Expect** doesn't know

about. Please find out what it is, and send this information back to me.

Ultrix 4.1 (at least the latest versions around here) considers timeouts of above 1000000 to be equivalent to 0.

Digital UNIX 4.0A (and probably other versions) refuses to allocate ptys if you define a SIGCHLD handler. See grantpt page for more info.

IRIX 6.0 does not handle pty permissions correctly so that if Expect attempts to allocate a pty previously used by someone else, it fails. Upgrade to IRIX 6.1.

Telnet (verified only under SunOS 4.1.2) hangs if TERM is not set. This is a problem under cron, at and in cgi scripts, which do not define TERM. Thus, you must set it explicitly - to what type is usually irrelevant. It just has to be set to something! The following probably suffices for most cases.

```
set env(TERM) vt100
```

Tip (verified only under BSDI BSD/OS 3.1 i386) hangs if SHELL and HOME are not set. This is a problem under cron, at and in cgi scripts, which do not define these environment variables. Thus, you must set them explicitly - to what type is usually irrelevant. It just has to be set to something! The following probably suffices for most cases.

```
set env(SHELL) /bin/sh
set env(HOME) /usr/bin
```

Some implementations of ptys are designed so that the kernel throws away any unread output after 10 to 15 seconds (actual number is implementation-dependent) after the process has closed the file descriptor. Thus **Expect** programs such as

```
spawn date
sleep 20
expect
```

will fail. To avoid this, invoke non-interactive programs with **exec** rather than **spawn**. While such situations are conceivable, in practice I have never encountered a situation in which the final output of a truly interactive program would be lost due to this behavior.

On the other hand, Cray UNICOS ptys throw away any unread output immediately after the process has closed the file descriptor. I have reported this to Cray and they are working on a fix.

Sometimes a delay is required between a prompt and a response, such as when a tty interface is changing UART settings or matching baud rates by looking for start/stop bits. Usually, all this is require is to sleep for a second or two. A more robust technique is to retry until the hardware is ready to receive input. The following example uses both strategies:

```
send "speed 9600\r";
sleep 1
expect {
   timeout {send "\r"; exp_continue}
   $prompt
}
```

trap −code will not work with any command that sits in Tcl's event loop, such as sleep. The problem is that in the event loop, Tcl discards the return codes from async event handlers. A workaround is to set a flag in the trap code. Then check the flag immediately after the command (i.e., sleep).

The expect_background command ignores −timeout arguments and has no concept of timeouts in general.

EXPECT HINTS

There are a couple of things about **Expect** that may be non-intuitive. This section attempts to address some of these things with a couple of suggestions.

A common expect problem is how to recognize shell prompts. Since these are customized differently by differently people and different shells, portably automating rlogin can be difficult without knowing the prompt. A reasonable convention is to have users store a regular expression describing their prompt (in particular, the end of it) in the environment variable EXPECT_PROMPT. Code like the following can be used. If EXPECT_PROMPT doesn't exist, the code still has a good chance of functioning correctly.

```
set prompt "(%|#|\\$) $"          ;# default prompt
catch {set prompt $env(EXPECT_PROMPT)}

expect −re $prompt
```

I encourage you to write **expect** patterns that include the end of whatever you expect to see. This avoids the possibility of answering a question before seeing the entire thing. In addition, while you may well be able to answer questions before seeing them entirely, if you answer early, your answer may appear echoed back in the middle of the question. In other words, the resulting dialogue will be correct but look scrambled.

Most prompts include a space character at the end. For example, the prompt from ftp is 'f', 't', 'p', '>' and <blank>. To match this prompt, you must account for each of these characters. It is a common mistake not to include the blank. Put the blank in explicitly.

If you use a pattern of the form X*, the * will match all the output received from the end of X to the last thing received. This sounds intuitive but can be somewhat confusing because the phrase "last thing received" can vary depending upon the speed of the computer and the processing of I/O both by the kernel and the device driver.

In particular, humans tend to see program output arriving in huge chunks (atomically) when in reality most programs produce output one line at a time. Assuming this is the case, the * in the pattern of the previous paragraph may only match the end of the current line even though there seems to be more, because at the time of the match that was all the output that had been received.

expect has no way of knowing that further output is coming unless your pattern specifically accounts for it.

Even depending on line-oriented buffering is unwise. Not only do programs rarely make promises about the type of buffering they do, but system indigestion can break output lines up so that lines break at seemingly random places. Thus, if you can express the last few characters of a prompt when writing patterns, it is wise to do so.

If you are waiting for a pattern in the last output of a program and the program emits something else instead, you will not be able to detect that with the **timeout** keyword. The reason is that**expect** will not timeout − instead it will get an **eof** indication. Use that instead. Even better, use both. That way if that line is ever moved around, you won't have to edit the line itself.

Newlines are usually converted to carriage return, linefeed sequences when output by the terminal driver. Thus, if you want a pattern that explicitly matches the two lines, from, say, printf("foo\nbar"), you should use the pattern "foo\r\nbar".

A similar translation occurs when reading from the user, via **expect_user**. In this case, when you press return, it will be translated to a newline. If**Expect** then passes that to a program which sets its terminal to raw mode (like telnet), there is going to be a problem, as the program expects a true return. (Some programs are actually forgiving in that they will automatically translate newlines to returns, but most don't.) Unfortunately, there is no way to find out that a program put its terminal into raw mode.

Rather than manually replacing newlines with returns, the solution is to use the command "stty raw", which will stop the translation. Note, however, that this means that you will no longer get the cooked line-editing features.

interact implicitly sets your terminal to raw mode so this problem will not arise then.

It is often useful to store passwords (or other private information) in **Expect** scripts. This is not recommended since anything that is stored on a computer is susceptible to being accessed by anyone. Thus, interactively prompting for passwords from a script is a smarter idea than embedding them literally. Nonetheless, sometimes such embedding is the only possibility.

Unfortunately, the UNIX file system has no direct way of creating scripts which are executable but unreadable. Systems which support setgid shell scripts may indirectly simulate this as follows:

Create the **Expect** script (that contains the secret data) as usual. Make its permissions be 750 (−rwxr−x−−−) and owned by a trusted group, i.e., a group which is allowed to read it. If necessary, create a new group for this purpose. Next, create a /bin/sh script with permissions 2751 (−rwxr−s−−x) owned by the same group as before.

The result is a script which may be executed (and read) by anyone. When invoked, it runs the **Expect** script.

SEE ALSO

Tcl(3), **libexpect**(3)

"Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs" by Don Libes, pp. 602, ISBN 1-56592-090-2, O'Reilly and Associates, 1995.

"expect: Curing Those Uncontrollable Fits of Interactivity" by Don Libes, Proceedings of the Summer 1990 USENIX Conference, Anaheim, California, June 11-15, 1990.

"Using **expect** to Automate System Administration Tasks" by Don Libes, Proceedings of the 1990 USENIX Large Installation Systems Administration Conference, Colorado Springs, Colorado, October 17-19, 1990.

"Tcl: An Embeddable Command Language" by John Ousterhout, Proceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990.

"expect: Scripts for Controlling Interactive Programs" by Don Libes, Computing Systems, Vol. 4, No. 2, University of California Press Journals, November 1991.

"Regression Testing and Conformance Testing Interactive Programs", by Don Libes, Proceedings of the Summer 1992 USENIX Conference, pp. 135-144, San Antonio, TX, June 12-15, 1992.

"Kibitz − Connecting Multiple Interactive Programs Together", by Don Libes, Software − Practice & Experience, John Wiley & Sons, West Sussex, England, Vol. 23, No. 5, May, 1993.

"A Debugger for Tcl Applications", by Don Libes, Proceedings of the 1993 Tcl/Tk Workshop, Berkeley, CA, June 10-11, 1993.

AUTHOR

Don Libes, National Institute of Standards and Technology

ACKNOWLEDGMENTS

Thanks to John Ousterhout for Tcl, and Scott Paisley for inspiration. Thanks to Rob Savoye for Expect's autoconfiguration code.

The HISTORY file documents much of the evolution of **expect**. It makes interesting reading and might give you further insight to this software. Thanks to the people mentioned in it who sent me bug fixes and gave other assistance.

Design and implementation of **Expect** was paid for in part by the U.S. government and is therefore in the

public domain. However the author and NIST would like credit if this program and documentation or portions of them are used.