The DejaGnu Testing Framework

for DejaGnu Version 1.1

Nov 1993

Rob Savoye

Cygnus Support
edited by Roland H. Pesch

Table of Contents

1 What is DejaGnu?

DejaGnu is a framework for testing other programs. Its purpose is to provide a single front end for all tests. Beyond this, DejaGnu offers several advantages for testing:

1. The flexibility and consistency of the DejaGnu framework make it easy to write tests for any program.
2. DejaGnu provides a layer of abstraction which allows you to write tests that are portable to any host or target where a program must be tested. For instance, a test for GDB can run (from any Unix based host) on any target architecture that DejaGnu supports. Currently DejaGnu runs tests on several single board computers, whose operating software ranges from just a boot monitor to a full-fledged, Unix-like realtime OS.
3. All tests have the same output format. This makes it easy to integrate testing into other software development processes.

DejaGnu is written in `expect`, which in turn uses *Tcl*—Tool command language.

Running tests requires two things: the testing framework, and the test suites themselves. Tests are usually written in `expect` using Tcl, but you can also use a Tcl script to run a test suite that is not based on `expect`. (`expect` script filenames conventionally use '`.exp`' as a suffix; for example, the main implementation of the DejaGnu test driver is in the file '`runtest.exp`'.)

1.1 Running existing tests

To run tests from an existing collection, first use `configure` as usual to set up the source directory containing the tests. Then try running

```
make check
```

If the `check` target exists, it usually saves you some trouble—for instance, it can set up any auxiliary programs or other files needed by the tests.

Once you have run '`make check`' to build any auxiliary files, you might want to call the test driver `runtest` directly to repeat the tests. You may also have to call `runtest` directly for test collections with no `check` target in the '`Makefile`'.

Typically, you must use two command-line options: '`--tool`', to specify which set of tests to run[1], and '`--srcdir`', to specify where to find test directories.

For example, if the directory '`gdb/testsuite`' contains a collection of DejaGnu tests for GDB, you can run them like this:

```
eg$ cd gdb/testsuite
eg$ runtest --tool gdb
```

Test output follows, ending with:

```
=== gdb Summary ===

# of expected passes 508
# of expected failures 103
/usr/latest/bin/gdb version 4.11.2 -nx
```

You can use the option '`--srcdir`' to point to some other directory containing a collection of tests:

```
eg$ runtest --tool gdb --srcdir /devo/gdb/testsuite
```

These examples assume a *native* configuration, where the same computer runs both `runtest` and the tests themselves. When you have a *cross* configuration, the tests run on a different computer, controlled by the host running `runtest`. In this situation, you need the option '`--name`' to specify the network address for the other computer:

```
eg$ runtest --tool gdb --name vx9.munist.com
```

If you always use the same option values, you can record them in a file called '`site.exp`', rather than typing them each time. See Section 3.2 "Defaults for `runtest` options," page 18.

By default, `runtest` prints only the names of the tests it runs, output from any tests that have unexpected results, and a summary showing how many tests passed and how many failed. To display output from all tests (whether or not they behave as expected), use the '`--all`' option. For more verbose output about processes being run, communication, and so on, use '`--verbose`'. To see even more output, use multiple '`--verbose`' options. See Chapter 2 "Using `runtest`," page 9, for a more detailed explanation of each `runtest` option.

Test output goes into two files in your current directory: summary output in '*`tool`*`.sum`', and detailed output in '*`tool`*`.log`'. (*`tool`* refers to the collection of tests; for example, after a run with '`--tool gdb`', look for

[1] '`--tool`' selects a particular suite of tests, *not* the name of the executable program to run. See Section 3.1 "Configuration dependent values," page 16, for information on the variables that you can use to specify the names of programs to run.

output files 'gdb.sum' and 'gdb.log'.) See Section 4.8 "The files DejaGnu writes," page 34.

1.2 What does a DejaGnu test look like?

Each DejaGnu test is an expect script; the tests vary widely in complexity, depending on the nature of the tool and the feature tested.

Here is a very simple GDB test—one of the simplest tests shipped with DejaGnu (extracted from 'gdb.t00/echo.exp'):[2]

```
# send a string to the GDB stdin:
send "echo Hello world!\n"

# inspect the GDB stdout for the correct reply,
# and determine whether the test passes or fails:
expect {
  -re "Hello world.*$prompt $"    { pass "Echo test" }
  -re "$prompt $"                 { fail "Echo test" }
  timeout                         { fail "(timeout) Echo test" }
  }
```

Though brief, this example is a complete test. It illustrates some of the main features of DejaGnu test scripts:

- The test case does not start the tested program (GDB in this case); all test scripts for interactive tools can assume the corresponding tool is running.
- Comments start with '#'.
- The main commands you use to control a tested program are send (to give it commands) and expect (to analyze its responses).
- The expect command uses a list of pairs; a pattern (regular expression if '-re' specified), followed by an action to run if the pattern matches output from the program. Only the action for the *first* matching pattern will execute.
- Test cases use the commands pass and fail to record the test outcome.

[2] More recent GDB tests use the 'gdb_test' procedure. An equivalent test using that procedure is 'gdb_test "echo Hello world!" "Hello world!"'

1.3 Design goals

DejaGnu grew out of the internal needs of Cygnus Support. Cygnus maintains and enhances a variety of free programs in many different environments, and we needed a testing tool that:

- is useful to developers while fixing bugs;
- automates running many tests during a software release process;
- is portable among a variety of host computers;
- supports cross-development testing;
- permits testing interactive programs, like GDB; and
- permits testing batch oriented programs, like GCC.

Some of the requirements proved challenging. For example, interactive programs do not lend themselves very well to automated testing. But all the requirements are important: for instance, it is imperative to make sure that GDB works as well when cross-debugging as it does in a native configuration.

Probably the greatest challenge was testing in a cross-development environment (which can be a real nightmare). Most cross-development environments are customized by each developer. Even when buying packaged boards from vendors there are many differences. The communication interfaces vary from a serial line to ethernet. DejaGnu was designed with a modular communication setup, so that each kind of communication can be added as required, and supported thereafter. Once a communication procedure is coded, any test can use it. Currently DejaGnu can use `rsh`, `rlogin`, `telnet`, `tip`, `kermit`, and `mondfe` for remote communications.

Julia Menapace first coined the term "Deja Gnu" to describe an earlier testing framework at Cygnus Support. When we replaced it with the Expect-based framework, it was like DejaGnu all over again...

1.4 A POSIX conforming test framework

DejaGnu conforms to the POSIX standard for test frameworks.

POSIX standard 1003.3 defines what a testing framework needs to provide, in order to permit the creation of POSIX conformance test suites. This standard is primarily oriented to running POSIX conformance tests, but its requirements also support testing of features not related to POSIX conformance. POSIX 1003.3 does not specify a particular testing framework, but at this time there is only one other POSIX conforming test framework: TET.[3]

The POSIX documentation refers to *assertions*. An assertion is a description of behavior. For example, if a standard says "The sun shall shine", a corresponding assertion might be "The sun is shining." A test based on this assertion would pass or fail depending on whether it is daytime or nighttime. It is important to note that the standard being tested is never 1003.3; the standard being tested is some other standard, for which the assertion was written.

As there is no test suite to test *testing frameworks* for POSIX 1003.3 conformance, verifying conformance to this standard is done by repeatedly reading the standard and experimenting. One of the main things 1003.3 does specify is the set of allowed output messages, and their definitions. Four messages are supported for a required feature of POSIX conforming systems, and a fifth for a conditional feature. DejaGnu supports the use of all five output messages; in this sense a test suite that uses exactly these messages can be considered POSIX conforming. These definitions specify the output of a test case:

`PASS`
: A test has succeeded. POSIX 1003.3 says that a known error can also be considered "success". Rather than use '`PASS`' in a confusing manner, DejaGnu recommends that tests restrict the meaning of '`PASS`' to a test case that has *not* reproduced the bug it was intended to capture. For known failures, DejaGnu tests can use the message '`XFAIL`' (or '`XPASS`', should a "known failure" start passing unexpectedly). (See Chapter 2 "Using `runtest`," page 9.)

  However, a test suite that may produce '`XFAIL`' will not conform to POSIX 1003.3. If you use DejaGnu to write POSIX test suites, you must use `PASS` ambiguously to mean either "no failure", or "known failure".

`FAIL`
: A test *has* produced the bug it was intended to capture. The '`FAIL`' message is based on the test case only. Other messages are used to indicate a failure of the framework.

`UNRESOLVED`
: A test produced indeterminate results. Usually, this means the test executed in an unexpected fashion; this outcome requires that a human being go over results, to determine if the test should have passed or failed. This message is also used for any test that requires human intervention because

[3] TET was created by Unisoft for a consortium comprised of X/Open, Unix International, and the Open Software Foundation.

it is beyond the abilities of the testing framework. Any unresolved test should resolved to 'PASS' or 'FAIL' before a test run can be considered finished.

Here are some of the ways a test may wind up 'UNRESOLVED':

- A test's execution is interrupted.
- A test does not produce a clear result. This is usually because there was an 'ERROR' from DejaGnu while processing the test, or because there were three or more 'WARNING' messages. Any 'WARNING' or 'ERROR' messages can invalidate the output of the test. This usually requires a human being to examine the output to determine what really happened—and to improve the test case.
- A test depends on a previous test, which fails.
- The test was set up incorrectly.

`UNTESTED` A test was not run. This is a placeholder, used when there is no real test case yet.

The only remaining output message left is intended to test features that are specified by the applicable POSIX standard as conditional:

`UNSUPPORTED`
There is no support for the tested case. This may mean that a conditional feature of an operating system, or of a compiler, is not implemented. DejaGnu also uses this message when a testing environment (often a "bare board" target) lacks basic support for compiling or running the test case. For example, a test for the system subroutine `gethostname` would never work on a target board running only a boot monitor.

DejaGnu uses the same output procedures to produce these messages for all test suites, and these procedures are already known to conform to POSIX 1003.3. Therefore, any DejaGnu test suite (so long as it avoids the `setup_xfail` procedure, and hence the 'XFAIL' and 'XPASS' messages) can be considered to conform to POSIX 1003.3.

1.5 Future directions

In the near future, there are two parallel directions for DejaGnu development. The first is to add support for more hosts and targets.

The second would permit testing programs with a more complex interface, whether text based or GUI based. Two components already exist: a Tcl based X window toolkit, and a terminal package for `expect`. Both of these could be merged into DejaGnu in a way that permits testing programs that run in each environment.

Meanwhile, we hope DejaGnu enables the creation of test suites for conformance to ANSI C and C++, to POSIX, and to other standards. We encourage you to make any test suites you create freely available, under the same terms as DejaGnu itself.

1.6 Tcl and Expect

Tcl was introduced in a paper by John K. Ousterhout at the 1990 Winter Usenix conference, *Tcl: An Embeddable Command Language*. That paper is included in PostScript form in the '`doc`' subdirectory of the Tcl distribution. See Appendix B "Tool Command Language Overview," page 87, for details on Tcl. The version of Tcl included in DejaGnu at this time is Tcl 6.7.

Don Libes introduced `expect` in his paper *expect: Curing Those Uncontrollable Fits of Interaction* at the 1990 Summer Usenix conference. The paper is included in PostScript form in the `expect` distribution (as are several other papers about `expect`). See Appendix A "Expect: Programmed Dialogue," page 51, for details on `expect`. The version of expect included in DejaGnu at this time is expect 4.7.6.1.

2 Using `runtest`

`runtest` is the executable test driver for DejaGnu. You can specify two kinds of things on the `runtest` command line: command line options, and Tcl variables for the test scripts. The options are listed alphabetically below. You can control the defaults for all `runtest` options; see Section 3.2 "Defaults for `runtest` options," page 18.

`runtest` returns an exit code of `1` if any test has an unexpected result; otherwise (if all tests pass or fail as expected) it returns `0` as the exit code.

`runtest` flags the outcome of each test as one of these cases. (See Section 1.4 "A POSIX conforming test framework," page 4, for a discussion of how POSIX specifies the meanings of these cases.)

`PASS`
: The most desirable outcome: the test succeeded, and was expected to succeed.

`XPASS`
: A pleasant kind of failure: a test was expected to fail, but succeeded. This may indicate progress; inspect the test case to determine whether you should amend it to stop expecting failure.

`FAIL`
: A test failed, although it was expected to succeed. This may indicate regress; inspect the test case and the failing software to locate the bug.

`XFAIL`
: A test failed, but it was expected to fail. This result indicates no change in a known bug. If a test fails because the operating system where the test runs lacks some facility required by the test, the outcome is `UNSUPPORTED` instead.

`UNRESOLVED`
: Output from a test requires manual inspection; the test suite could not automatically determine the outcome. For example, your tests can report this outcome is when a test does not complete as expected.

`UNTESTED`
: A test case is not yet complete, and in particular cannot yet produce a `PASS` or `FAIL`. You can also use this outcome in dummy "tests" that note explicitly the absence of a real test case for a particular property.

`UNSUPPORTED`
: A test depends on a conditionally available feature that does not exist (in the configured testing environment). For example, you can use this outcome to report on a test case that does not work on a particular target because its operating system support does not include a required subroutine.

runtest may also display the following messages:

ERROR
: Indicates a major problem (detected by the test case itself) in running the test. This is usually an unrecoverable error, such as a missing file or loss of communication to the target. (POSIX test suites should not emit this message; use UNSUPPORTED, UNTESTED, or UNRESOLVED instead, as appropriate.)

WARNING
: Indicates a possible problem in running the test. Usually warnings correspond to recoverable errors, or display an important message about the following tests.

This is the full set of command line options that runtest recognizes. Arguments may be abbreviated to the shortest unique string.

```
runtest --tool tool  [ testsuite.exp ... ]
[ testsuite.exp="testfile1 ..." ]
[ tclvar=value... ]
[ --all ]  [ --baud baud-rate ]  [ --connect type ]
[ --debug ]  [ --help ]  [ --host string ]
[ --mail "name ..." ]  [ --name string ]
[ --name name ]  [ --outdir path ]
[ --objdir path ]  [ --reboot ]
[ --srcdir path ]  [ --strace n ]
[ --target string ]
[ -v | --verbose ]  [ -V | --version ]  [ --Dn ]
```

--tool *tool*
: *tool* specifies what set of tests to run, and what initialization module to use. *tool* is used *only* for these two purposes: it is *not* used to name the executable program to test. Executable tool names (and paths) are recorded in 'site.exp' (see Section 3.1 "Configuration dependent values," page 16), and you can override them by specifying Tcl variables on the command line.

  For example, including '--tool gcc' on the runtest command line runs tests from all test subdirectories whose names match 'gcc.*', and uses one of the initialization modules named 'config/*-gcc.exp'. To specify the name of the compiler (perhaps as an alternative path to what runtest would use by default), use 'GCC=*binname*' on the runtest command line.

testsuite.exp . . .
: Specify the names of testsuites to run. By default, runtest runs all tests for the tool, but you can restrict it to particular testsuites by giving the names of the '.exp' expect scripts that control them.

testsuite.exp may not include path information; use plain filenames.

testfile.exp="testfile1 . . ."
: Specify a subset of tests in a suite to run. For compiler or assembler tests, which often use a single '.exp' script covering many different source files, this option allows you to further restrict the tests by listing particular source files to compile. Some tools even support wildcards here. The wildcards supported depend upon the tool, but typically they are ?, *, and [chars].

tclvar=*value*
: You can define Tcl variables for use by your test scripts in the same style used with make for environment variables. For example, 'runtest GDB=gdb.old' defines a variable called 'GDB'; when your scripts refer to '$GDB' in this run, they use the value 'gdb.old'.

The default Tcl variables used for most tools are defined in the main DejaGnu Makefile; their values are captured in the 'site.exp' file. See Section 3.1 "Configuration dependent values," page 16.

--all
: Display all test output. By default, runtest shows only the output of tests that produce unexpected results; that is, tests with status 'FAIL' (unexpected failure), 'XPASS' (unexpected success), or 'ERROR' (a severe error in the test case itself). Specify '--all' to see output for tests with status 'PASS' (success, as expected) 'XFAIL' (failure, as expected), or 'WARNING' (minor error in the test case itself).

--baud *baud-rate*
-b *baud-rate*
: Set the default baud rate to something other than 9600. (Some serial interface programs, like tip, use a separate initialization file instead of this value.)

--connect *type*
: Connect to a target testing environment as specified by *type*, if the target is not the computer running runtest. For example, use '--connect' to change the program used to connect to a "bare board" boot monitor. The choices for *type* in the DejaGnu 1.0 distribution are 'rlogin', 'telnet', 'rsh', 'tip', 'kermit', and 'mondfe'.

The default for this option depends on the configuration (see Section 4.6 "Remote targets supported," page 30). The default is chosen to be the most convenient communication method available, but often other alternatives work as well; you may find it useful to try alternative connect methods if you suspect a communication problem with your testing target.

`--debug`
: Turns on the `expect` internal debugging output. Debugging output is displayed as part of the `runtest` output, and logged to a file called '`dbg.log`'. The extra debugging output does *not* appear on standard output, unless the verbose level is greater than 2 (for instance, to see debug output immediately, specify '`--debug -v -v`'). The debugging output shows all attempts at matching the test output of the tool with the scripted patterns describing expected output. The output generated with '`--strace`' also goes into '`dbg.log`'.

`--help`
`-he`
: Prints out a short summary of the `runtest` options, then exits (even if you also specify other options).

`--host` *`string`*
: *`string`* is a full configuration "triple" name as used by `configure`. Use this option to override the default string recorded by your configuration's choice of host. This choice does not change how anything is actually configured; it affects *only* DejaGnu procedures that compare the host string with particular values. The procedures `ishost`, `istarget`, `isnative`, and `setup_xfail` are affected by '`--host`'.

`--mail "`*`name1 name2`*` . . ."`
`--mail` *`name`*
: Send the summary testing output to the specified list, via email. The quotes are only needed if you supply more than one name with a single '`--mail`' option.

`--name` *`name`*
: *`name`* is a name for the particular testing target machine (for cross testing). If the testing target has IP network support (for example, `RPC` or `NFS`), this is the network name for the target itself. (*`name`* is *not the configuration string* you specify as a target with `configure`; the '`--name`' option names a particular target, rather than describing a class of targets.) For targets that connect in other ways, the meaning of the *`name`* string depends on the connection method. See Section 4.6 "Remote targets supported," page 30.

`--name` *`string`*
: Specify a network name of testing target or its host. The particular names that are meaningful with '`--name`' will depend on your site configuration, and on the connection protocol: for example, `tip` connections require names from a serial line configuration file (usually called '`/etc/remote`'), while `telnet` connections use IP hostnames.

`--objdir` *`path`*
: Use *`path`* as the top directory containing any auxiliary compiled test code. This defaults to '`.`'. Use this option to locate pre-compiled test code. You can normally prepare any auxiliary files needed with `make`.

`--outdir` *`path`*
: Write output logs in directory *`path`*. The default is '`.`', the directory where you start `runtest`. This option affects only the summary and the detailed log files '*`tool`*`.sum`' and '*`tool`*`.log`'. The DejaGnu debug log '`dbg.log`' always appears (when requested) in the local directory.

`--reboot`
: Reboot the target board when `runtest` initializes. Usually, when running tests on a separate target board, it is safer to reboot the target to be certain of its state. However, when developing test scripts, rebooting takes a lot of time.

`--srcdir` *`path`*
: Use *`path`* as the top directory for test scripts to run. `runtest` looks in this directory for any subdirectory whose name begins with the toolname (specified with '`--tool`'). For instance, with '`--tool gdb`', `runtest` uses tests in subdirectories '`gdb.*`' (with the usual shell-like filename expansion). If you do not use '`--srcdir`', `runtest` looks for test directories under the current working directory.

`--strace` *`n`*
: Turn on internal tracing for `expect`, to *`n`* levels deep. By adjusting the level, you can control the extent to which your output expands multi-level Tcl statements. This allows you to ignore some levels of `case` or `if` statements. Each procedure call or control structure counts as one "level".

: The output is recorded in the same file, '`dbg.log`', used for output from '`--debug`'.

`--target string`
: Use this option to override the default setting (running native tests). *`string`* is a full configuration "triple" name[1] as used by `configure`. This option changes the configuration `runtest` uses for the default tool names, and other setup information. See section "Using `configure`" in *Cygnus configure*, for details about `configure` names.

`--verbose`
`-v`
: Turns on more output. Repeating this option increases the amount of output displayed. Level one ('`-v`') is simply test output. Level two ('`-v -v`') shows messages on options, configuration, and process control. Verbose messages appear in the detailed ('`*.log`') log file, but not in the summary ('`*.sum`') log file.

`--version`
`-V`
: Prints out the version numbers of DejaGnu, `expect` and Tcl, and exits without running any tests.

`-D0`
`-D1`
: Start the internal Tcl debugger. The Tcl debugger supports breakpoints, single stepping, and other common debugging activities. (See *A Debugger for Tcl Applications* by Don Libes.[2])

  If you specify '`-D1`', the `expect` shell stops at a breakpoint as soon as DejaGnu invokes it.

  If you specify '`-D0`', DejaGnu starts as usual, but you can enter the debugger by sending an interrupt (e.g. by typing `C-C`).

---

[1] Configuration triples have the form '*`cpu-vendor-os`*'.

[2] Distributed in PostScript form with `expect` as the file '`expect/tcl-debug.ps`'.

3 Setting `runtest` defaults

The site configuration file, 'site.exp', captures configuration-dependent values and propagates them to the DejaGnu test environment using Tcl variables. This ties the DejaGnu test scripts into the `configure` and `make` programs.

DejaGnu supports more than one 'site.exp' file. The multiple instances of 'site.exp' are loaded in a fixed order built into DejaGnu (the more local last).

1. There is a "master" 'site.exp', capturing configuration values that apply to DejaGnu across the board, in each configuration-specific subdirectory of the DejaGnu library directory. `runtest` loads these values first. This master configuration file is created when DejaGnu is installed. See Appendix C "Configuring and Installing DejaGnu," page 149. The master 'site.exp' can contain the default values for all targets supported by DejaGnu.
2. You can also define another global configuration file, and identify it by setting the environment variable `DEJAGNU` to the name of the file. Anything in the file '$DEJAGNU' overrides the values in the installed master configuration file.
3. Any directory containing a configured test suite also has a 'site.exp', capturing configuration values specific to the tool under test. Since `runtest` loads these values last, the individual test configuration can either rely on and use, or override, any of the global values from the "master" 'site.exp'.

   You can usually generate or update the test-suite 'site.exp' by typing 'make site.exp' in the test suite directory, after the test suite is configured.

You can further override the default values in a user-editable section of any 'site.exp', or by setting variables on the `runtest` command line.

All 'site.exp' files have two sections, separated by comment text. The first section starts with:

```
## these variables are automatically generated by make ##
# Do not edit here. If you wish to override these values
# add them to the last section
```

Do not edit this first section; if you do, your changes are replaced next time you run `make`. The first section ends with this line:

```
## All variables above are generated by configure. Do Not Edit ##
```

The first section is the part that is generated by `make`. It is essentially a collection of Tcl variable definitions based on 'Makefile' environment variables. Since they are generated by `make`, they contain the values as specified by `configure`. (You can also customize these values by using the '--site' option to `configure`.) In particular, this section contains the 'Makefile' variables for host and target configuration data.

In the second section, you can override any default values (locally to DejaGnu) for all the variables. The second section can also contain your preferred defaults for all the command line options to `runtest`. This allows you to easily customize `runtest` for your preferences in each configured test-suite tree, so that you need not type options repeatedly on the command line. (The second section may also be empty, if you do not wish to override any defaults.)

It is usually more convenient to keep these "manual overrides" in the 'site.exp' local to each test directory, rather than in the "master" 'site.exp' in the DejaGnu library.

To override the values captured in the first part of 'site.exp' by `configure` and `make`, simply set your own value for any of these variables *in the second part* of 'site.exp'. Set the variables using the Tcl `set` command; for example, to run tests on an experimental version of GCC, the second part of your 'site.exp' might include:

```
set CC "/working/devo/gcc/xgcc"
```

3.1 Configuration dependent values

The default values depend on your configuration. The defaults shown are those built into the DejaGnu distribution, which will appear only if your configuration does not change them. (For example, in cross configurations, program names often reflect the target type.)

This example is taken from the template for the global configuration file installed as the master 'site.exp'. You can use the same format (a Tcl `case` statement) for any DejaGnu configuration file.

```
#
# Set defaults for all known targets
#
case "$target_alias" in {
    { "native" } {
set target_abbrev unix
set connectmode ""
set targetname  ""
set netport     ""
set serialport  ""
set baud        ""
    }
    { "m68k-*bug-*" "m68k-mvme135-*" } {
        # Motorola MVME135 board running Bug monitor
set target_abbrev bug
set connectmode "tip"
set targetname  "s8"
set serialport  "/dev/ttyr8"
set netport     "wharfrat:20008"
set baud        "9600"
set CFLAGS      "-Tmvme135.ld"
set CXXFLAGS    "-Tmvme135.ld"
     }
}
```

`$target_alias`
: The value of `target_alias` is set by `configure`. This DejaGnu configuration file uses it to choose the default values for a particular target machine.

`m68k-*bug-*`
`m68k-mvme135-*`
: These are examples of configuration strings to be compared against '`$target_alias`'. The first of these strings matches the configuration named '`m68k-abug-coff`'.

`target_abbrev`
: The abbreviation used to construct the tool initialization filename. This only exists because many systems have restrictions on how long a file name can be.

`connectmode`
: The name of a Tcl procedure which manages connecting to the target board. See Section 3.2 "Defaults for runtest options," page 18, for the list of meaningful values.

`targetname`
: A string passed to the '`$connectmode`' procedure to identify a particular target machine. In this example, '`s8`' is an entry in /etc/remote, used by `tip`.

serialport
: Which serial port to use, if '$connectmode' specifies a serial connection procedure.

netport
: Which TCP port to use, if a target is available over an internet (for example, on a terminal server), rather than through a standard serial communications line.

baud
: Speed (in bps) used to connect with a target via a serial line.

CFLAGS
: The default list of options passed to the C compiler, whenever it is used in tests. The default value is '-g'.

CXXFLAGS
: The default list of options passed to the C++ compiler, whenever it is used in tests. The default value is "-g".

The following variables, usually in the local 'site.exp' (in test directories), extend the standard set of configuration variables. You should never redefine these; they permit runtest to determine its configuration at runtime. The examples shown below are from a Sun-4 native configuration.

host_triplet
: The full configuration name of the host computer where DejaGnu runs; this is the default corresponding to the '--host' option to runtest. "sparc-sun-sunos4.1.1" describes a particular level of Sun4, for example.

target_triplet
: The full configuration name of the target machine where the tests run; in a native configuration, the same value as 'host_triplet'.

3.2 Defaults for runtest options

In the user editable second section of 'site.exp', you can not only override the configuration variables captured in the first section, but also specify default values for all the runtest command line options. Save for '--debug', '--help', and '--version', each command line option has an associated Tcl variable. Use the Tcl set command to specify a new default value (as for the configuration variables). The following table describes the correspondence between command line options and variables you can set in 'site.exp'. See Chapter 2 "Running the Tests," page 9, for explanations of the command-line options.

runtest option	*Tcl variable*	*description*
--all	all_flag	display all test results if set
--baud	baud	set the default baud rate to something other than 9600.
--connect	connectmode	'rlogin', 'telnet', 'rsh', 'kermit', 'tip', or 'mondfe'
--mail	mailing_list	address list for mailing test output
--name	targetname	network name of testing target or its host
--outdir	outdir	directory for '*tool*.sum' and '*tool*.log'
--objdir	objdir	directory for compiled binaries
--reboot	reboot	reboot the target if set to "1"; do not reboot if set to "0" (the default)
--srcdir	srcdir	directory of test subdirectories
--strace	tracelevel	a number: Tcl trace depth
--tool	tool	name of tool to test; identifies init, test subdir
--verbose	verbose	verbosity level. As option, use multiple times; as variable, set a number, 0 or greater

4 The DejaGnu Implementation

DejaGnu is entirely written in `expect`, which uses Tcl as a command language. `expect` serves as a very programmable shell; you can run any program, as with the usual Unix command shells—but once the program is started, your `expect` script has fully programmable control of its input and output. This does not just apply to the programs under test; `expect` can also run any auxiliary program, such as `diff` or `sh`, with full control over its input and output.

DejaGnu itself is merely a framework for the set of test suites distributed separately for each GNU tool. Future releases of GNU tools will include even more tests, developed throughout the free software community.

`runtest` is the glue to tie together and manage the test scripts. The `runtest` program is actually a simple Bourne shell script that locates a copy of the `expect` shell and then starts the main Tcl code, `runtest.exp`. `runtest.exp` itself has these essential functions:

1. Parse the command line options, load the library files, and load the default configuration files.
2. Locating the individual test scripts. `runtest.exp` locates the tests by exploiting a straightforward naming convention based on the string you specify with the '`--tool`' option.
3. Providing an extended test environment, by defining additional Tcl procedures beyond those already in `expect`.
4. Locating target-dependent functions, to standardize the test environment across a wide variety of test platforms.

4.1 Conventions for using tool names

DejaGnu uses '`$tool`', the name of the tool under test, to tie together the testing configuration in a straightforward but flexible way.

'`$tool`' is *not* used to invoke the tool, since sites that run multiple configurations of a particular tool often call each configuration by a different name. `runtest` uses the configuration-dependent variables captured in '`site.exp`' to determine how to call each tool.

`runtest` uses tool names to find directories containing tests. `runtest` scans the source directory (specified with `--srcdir`) for all directories whose names start with the tool name followed by a period. For instance, directories that start with '`g++.`' contain G++ tests. To add a new test, just put it in any directory (create an entirely new directory, if you wish) whose name follows this convention.

A test is any file in an appropriately named subdirectory whose name ends in '.exp' (the conventional way of naming expect scripts). These simple naming conventions make it as simple as possible to install new tests: all you must do is put the test in the right directory.

runtest sorts the tests in each subdirectory by name (using the Tcl lsort command) and runs them in the resulting order.

4.2 Initialization module

The initialization module (or "init file") has two purposes: to provide tool and target dependent procedures, and to start up an interactive tool to the point where it is ready to operate. The latter includes establishing communications with the target. All the tests for interactive programs assume that the tool is already running and communicating. Initialization modules for non-interactive programs may only need to supply the support functions.

Each test suite directory must contain (in its 'config' subdirectory) a separate initialization module for each target. The appropriate init file is identified by name: it must be named '$target_abbrev.exp', where target_abbrev is the Tcl variable recording the abbreviated configuration name for the target operating system.

If the test suite contains tests for more than one tool, it can either use a single init file for all the tools, or segregate the init files by tool using a more elaborate naming convention: '$target_abbrev-$tool.exp', where tool is the Tcl variable identifying tests for a particular tool. It is better to avoid this convention if possible, since it may lead to unacceptable constraints on tool names due to systems with low limits on file-name length. It is more portable to segregate tests for different tools into different test-suite directories, instead.

At the beginning of the init file, you must first determine the proper executable name of the tool to execute, since the actual name of the tool to be tested my vary from system to system. Here's an example for the GNU C compiler.

```
global CC
if ![info exists CC] then {
    set CC [transform gcc]
}

global CFLAGS
if ![info exists CFLAGS] then {
    set CFLAGS ""
}
```

It is always a good idea to first check the variable, and only set it if it has not yet been defined. Often the proper value of CC is set on the command line that invokes 'runtest'.

The transform procedure takes as its argument the native name of a tool (such as 'gcc' for the compiler), and returns the name as configured for that tool in the current installation. (For example, a cross-compiling version of GNU CC that generates MIPS code may be installed with a name like mips-idt-ecoff-gcc.)

In a test running native, writing the Tcl code for initialization is usually quite simple. For cross configurations, however, more elaborate instructions are usually needed to describe how to talk to a remote target.

Each initialization module defines up to four procedures with standard names and purposes. The names of these procedures begin with '$tool', the string that identifies tests for a particular tool: $tool_start, $tool_load, $tool_exit, and $tool_version. For example, the start procedure for GDB is called gdb_start. (Since start procedures are used differently for batch and interactive tools, however, runtest itself never calls the start procedure. Init files for interactive tools are expected to end by running the start procedure.)

The initialization module is also a good place to call load_lib to get any collections of utility procedures meant for a family of test cases, and to set up default values for any additional Tcl variables needed for a specific set of tests.

See Section 4.5 "Target dependent procedures," page 29, for full descriptions of these procedures.

4.3 DejaGnu procedures

DejaGnu provides these Tcl procedures for use in test scripts. You can also use any standard expect or Tcl function.

See Section 1.4 "A POSIX conforming test framework," page 4, for more detailed explanations of the test outcomes ('FAIL', 'PASS', 'UNTESTED', 'UNRESOLVED', 'UNSUPPORTED').

perror "*string*"

Declares a severe error in the testing framework itself. perror writes in the log files a message beginning with 'ERROR', appending the argument *string*.

As a side effect, perror also changes the effect of the next pass or fail command: the test outcome becomes 'UNRESOLVED', since an automatic 'PASS' or 'FAIL' cannot be trusted after a severe error in the test framework.

`warning "string"`
: Declares detection of a minor error in the test case itself. `warning` writes in the log files a message beginning with 'WARNING', appending the argument *string*. Use `warning` rather than `error` for cases (such as communication failure to be followed by a retry) where the test case can recover from the error.

  As a side effect, three or more calls to `warning` in a single test case also change the effect of the next `pass` or `fail` command: the test outcome becomes 'UNRESOLVED', since an automatic 'PASS' or 'FAIL' may not be trustworthy after many warnings.

`pass "string"`
: Declares a test to have passed. `pass` writes in the log files a message beginning with 'PASS' (or XPASS, if failure was expected), appending the argument *string*.

`fail "string"`
: Declares a test to have failed. `fail` writes in the log files a message beginning with 'FAIL' (or XFAIL, if failure was expected), appending the argument *string*.

`unresolved "string"`
: Declares a test to have an unresolved outcome. `unresolved` writes in the log file a message beginning with 'UNRESOLVED', appending the argument *string*. This usually means the test did not execute as expected, and a human being must go over results to determine if it passed or failed (and to improve the test case).

`untested "string"`
: Declares a test was not run. `untested` writes in the log file a message beginning with 'UNTESTED', appending the argument *string*. For example, you might use this in a dummy test whose only role is to record that a test does not yet exist for some feature.

`unsupported "string"`
: Declares that a test case depends on some facility that does not exist in the testing environment. `unsupported` writes in the log file a message beginning with 'UNSUPPORTED', appending the argument *string*.

`transform "toolname"`
: Generates a string for the name of a tool as it was configured and installed, given its native name (as the argument *toolname*). This makes the assumption that all tools are

installed using the same naming conventions: it extrapolates from the invocation name for '`runtest`'. For example, if you call `runtest` as '`m68k-vxworks-runtest`', the result of '`transform "gcc"`' is '`m68k-vxworks-gcc`'.

`ishost "`*`host`*`"`
: Tests for a particular *host* environment. If the currently configured host matches the argument string, the result is `1`; otherwise the result is `0`. *`host`* must be a full three-part `configure` host name; in particular, you may not use the shorter nicknames supported by `configure` (but you can use wildcard characters, using shell syntax, to specify sets of names).

`istarget "`*`target`*`"`
: Tests for a particular *target* environment. If the currently configured target matches the argument string, the result is `1`; otherwise the result is `0`. *`target`* must be a full three-part `configure` target name; in particular, you may not use the shorter nicknames supported by `configure` (but you can use wildcard characters, using shell syntax, to specify sets of names).

`isnative`
: Tests whether the current configuration has the same host and target. When it runs in a *native* configuration this procedure returns a `1`; otherwise it returns a `0`.

`load_lib "`*`library-file`*`"`
: Loads the file *`library-file`* by searching a fixed path built into `runtest`. If DejaGnu has been installed, it looks in a path starting with the installed library directory. If you are running DejaGnu directly from a source directory, without first running '`make install`', this path defaults to the current directory. In either case, it then looks in the current directory for a directory called `lib`. If there are duplicate definitions, the last one loaded takes precedence over the earlier ones.

`setup_xfail "`*`config`*` [`*`bugid`*`]"`
: Declares that the test is expected to fail on a particular set of configurations. The *`config`* argument must be a list of full three-part `configure` target name; in particular, you may not use the shorter nicknames supported by `configure` (but you can use the common shell wildcard characters to specify sets of names). The *`bugid`* argument is optional, and used only in the logging file output; use it as a link to a bug-tracking system such as GNATS (see section "Overview" in *Tracking Bugs With GNATS*).

Once you use `setup_xfail`, the `fail` and `pass` procedures produce the messages 'XFAIL' and 'XPASS' respectively, allowing you to distinguish expected failures (and unexpected success!) from other test outcomes.

Warning: you must clear the expected failure after using `setup_xfail` in a test case. Any call to `pass` or `fail` clears the expected failure implicitly; if the test has some other outcome, e.g. an error, you can call `clear_xfail` to clear the expected failure explicitly. Otherwise, the expected-failure declaration applies to whatever test runs next, leading to surprising results.

`clear_xfail` *config*

Cancel an expected failure (previously declared with `setup_xfail`) for a particular set of configurations. The *config* argument is a list of configuration target names. It is only necessary to call `clear_xfail` if a test case ends without calling either `pass` or `fail`, after calling `setup_xfail`.

`verbose` [-n] [--] "*string*" *number*

Test cases can use this function to issue helpful messages depending on the number of '`--verbose`' options on the `runtest` command line. It prints *string* if the value of the variable `verbose` is higher than or equal to the optional *number*. The default value for *number* is 1. Use the optional '-n' argument to print *string* without a trailing newline. Use the optional '--' argument if *string* begins with "-".

4.4 Library procedures

Certain other general-purpose procedures are kept as utility collections in the '`lib`' subdirectory. These files are not used by all tests, but are used (via the Tcl command `source`) often enough to warrant a common definition.

'`lib/remote.exp`' defines these functions, for establishing and managing communications:

Procedures to establish a connection: Each of these procedures tries to establish the connection up to three times before returning. Warnings (if retries will continue) or errors (if the attempt is abandoned) report on communication failures. The result for any of these procedures is either `-1`, when the connection cannot be established, or the spawn ID returned by the `expect` command `spawn`.

`telnet` *hostname*
`rlogin` *hostname*
`rsh` *hostname*
: *IP network procedures.* *hostname* refers to the IP address or name (for example, an entry in '/etc/hosts') for this target. The procedure names reflect the Unix utility used to establish a connection.

`tip` *port*
: *Serial line procedure.* Connect using the Unix utility `tip`. *port* must be a name from the `tip` configuration file '/etc/remote'. Often, this is called 'hardwire', or something like 'ttya'. This file holds all the configuration data for the serial port.

`kermit` *port bps*
: *Serial line procedure.* Connect using the program `kermit`. *port* is the device name, e.g. '/dev/ttyb'. *bps* is the line speed to use (in bits per second) for the connection.

Procedures to manage a connection:

`tip_download` *spawnid file*
: Download '*file*' to the process *spawnid* (the value returned when the connection was established), using the `~put` command under `tip`. Most often used for single board computers that require downloading programs in ASCII S-records. Returns `1` if an error occurs, `0` otherwise.

`exit_remote_shell` *spawnid*
: Exits a remote process started by any of the connection procedures. *spawnid* is the result of the connection procedure that started the remote process.

`download` *file* [*spawnid*]
: After you establish a connection to a target, you can download programs using this command. `download` reads in *file* (object code in S-record format) and writes it to the device controlling this *spawnid*. (From the point of view of the target, the S-record file comes in via standard input.)

  If you have more than one target active, you can use the optional argument *spawnid* to specify an alternative target (the default is the most recently established *spawnid*.)

'lib/utils.exp' defines these utility procedures:

`getdirs` *`dir`*
`getdirs` *`dir pattern`*
: Returns a list of all the directories in the single directory *`dir`* that match *`pattern`*. If you do not specify *`pattern`*, `getdirs` assumes '`*`'. You may use the common shell wildcard characters in *`pattern`*.

`find` *`dir pattern`*
: Search for files whose names match *`pattern`* (using shell wildcard characters for filename expansion). Search subdirectories recursively, starting at *`dir`*. The result is the list of files whose names match; if no files match, the result is empty. Filenames in the result include all intervening subdirectory names.

`which` *`binary`*
: Searches the execution path for an executable file *`binary`*, like the the BSD `which` utility. This procedure uses the shell environment variable '`PATH`'. It returns `0` if the binary is not in the path, or if there is no '`PATH`' environment variable. If *`binary`* is in the path, it returns the full path to *`binary`*.

`grep` *`filename regexp`*

`grep` *`filename regexp`* `line`
: Search the file called *`filename`* (a fully specified path) for lines that contain a match for regular expression *`regexp`*. The result is a list of all the lines that match. If no lines match, the result is an empty string. Specify *`regexp`* using the standard regular expression style used by the Unix utility program `grep`.

  Use the optional third argument '`line`' to start lines in the result with the line number in *`filename`*. (This argument is simply an option flag; type it just as shown—'`line`'.)

`prune` *`list pattern`*
: Remove elements of the Tcl list *`list`*. Elements are fields delimited by spaces. The result is a copy of *`list`*, without any elements that match *`pattern`*. You can use the common shell wildcard characters to specify *`pattern`*.

`runtest_file_p` *`runtests testcase`*
: Search *`runtests`* for *`testcase`* and return 1 if found, 0 if not. *`runtests`* is a list of two elements. The first is the pathname of the testsuite expect script running. The second is a copy of what was on the right side of the = if '`foo.exp="..."`' was specified, or an empty string if no such argument is present. This is used by tools like compilers where each testcase is a file.

`prune_system_crud` *system text*

For system *system*, delete text the host or target operating system might issue that will interfere with pattern matching of program output in *text*. An example is the message that is printed if a shared library is out of date.

Most tools also have a library file associated with the tool, containing procedures found useful by test developers. These are especially helpful when the tests span several subdirectories.

4.5 Target dependent procedures

Each combination of target and tool requires some target-dependent procedures. The names of these procedures have a common form: the tool name, followed by an underbar '_', and finally a suffix describing the procedure's purpose. For example, a procedure to extract the version from GDB is called 'gdb_version'. See Section 4.2 "Initialization Module," page 22, for a discussion of how DejaGnu arranges to find the right procedures for each target.

`runtest` itself calls only two of these procedures, *tool*_exit and *tool*_version; these procedures use no arguments.

The other two procedures, *tool*_start and *tool*_load, are only called by the test suites themselves (or by testsuite-specific initialization code); they may take arguments or not, depending on the conventions used within each test suite.

*tool*_start

Starts a particular tool. For an interactive tool, *tool*_start starts and initializes the tool, leaving the tool up and running for the test cases; an example is `gdb_start`, the start function for GDB. For a batch oriented tool, *tool*_start is optional; the recommended convention is to let *tool*_start run the tool, leaving the output in a variable called `comp_output`. Test scripts can then analyze '`$comp_output`' to determine the test results. An example of this second kind of start function is `gcc_start`, the start function for GCC.

`runtest` itself *does not call* *tool*_start. The initialization module '*tool*_init.exp' must call *tool*_start for interactive tools; for batch-oriented tools, each individual test script calls *tool*_start (or makes other arrangements to run the tool).

`tool_load`
: Loads something into a tool. For an interactive tool, this conditions the tool for a particular test case; for example, `gdb_load` loads a new executable file into the debugger. For batch oriented tools, *tool*`_load` may do nothing—though, for example, the GCC support uses `gcc_load` to load and run a binary on the target environment. Conventionally, *tool*`_load` leaves the output of any program it runs in a variable called '`exec_output`'. Writing *tool*`_load` can be the most complex part of extending DejaGnu to a new tool or a new target, if it requires much communication coding or file downloading.

  Test scripts call *tool*`_load`.

tool`_exit`
: Cleans up (if necessary) before `runtest` exits. For interactive tools, this usually ends the interactive session. You can also use *tool*`_exit` to remove any temporary files left over from the tests.

  `runtest` calls *tool*`_exit`.

tool`_version`
: Prints the version label and number for *tool*. This is called by the DejaGnu procedure that prints the final summary report. The output should consist of the full path name used for the tested tool, and its version number.

  `runtest` calls *tool*`_version`.

The usual convention for return codes from any of these procedures (although it is not required by `runtest`) is to return `0` if the procedure succeeded, `1` if it failed, and `-1` if there was a communication error.

4.6 Remote targets supported

The DejaGnu distribution includes support for the following remote targets. You can set the target name and the connect mode in the '`site.exp`' file (using the Tcl variables '`targetname`' and '`connectmode`', respectively), or on the `runtest` command line (using '`--name`' and '`--connect`').

AMD **29000, with UDI protocol**
: Configure DejaGnu for target '`a29k-amd-udi`'. (Cygnus `configure` also recognizes the abbreviation '`udi29k`'.) Then, to run tests, use the `runtest` target name to specify whether you want to use a simulator, or a particular hardware board. The particular string to use with '`--name`' will depend on your UDI setup file, '`udi_soc`' (if '`udi_soc`' is not in your working

directory, the environment variable 'UDICONF' should contain a path to this file). For example, if your UDI setup file includes these lines:

```
iss   AF_UNIX  *   isstip -r /home/gnu/29k/src/osboot/sim/osboot
mon   AF_UNIX  *   montip -t serial -baud 9600 -com /dev/ttyb
```

You can use '--name iss' to run tests on the simulator, and '--name mon' to run tests on the 29K hardware. See the manufacturer's manuals for more information on UDI and 'udi_soc'.

The default connect protocol is 'mondfe' with either back end. mondfe is the only shell DejaGnu supports for UDI targets. mondfe is an AMD specific monitor program freely available from AMD.

Warning: This target requires GDB version 4.7.2 (or greater). Earlier versions of GDB do not fully support the load command on this target, so DejaGnu has no way to load executable files from the debugger.

Motorola 680x0 boards, a.out or COFF object format

Configure DejaGnu for any remote target matching 'm68k-*'.

Warning: Most 'm68k-*' configurations run all tests only for native testing (when the target is the same as the host). When you specify most of these targets for a cross configuration, you will only be able to use tests that run completely within the host (for example, tests of the binary utilities such as the archiver; or compiler tests that only generate code rather than running it).

To run a.out or COFF binaries on a remote M68K, you must configure DejaGnu for a particular target board. 'm68k-abug' is an example. (In general for an embedded environment, because it does not have absolute addresses, a.out is not a good choice for output format in any case; most often S-records or Hex-32 are used instead.)

Motorola 68K MVME 135 board running ABug boot monitor

Configure for 'm68k-abug-aout' or 'm68k-abug-coff' (as a target). This boot monitor can only download S-records; therefore, the DejaGnu tests for this environment require a linker command script to convert either output format to S-records, setting the default addresses for .text, .bss, and .data.

With this configuration, the default for '--connect' is 'tip'. 'tip' is the only communications protocol supported for connecting to 'm68k-abug-*' targets. 'tip' uses an ASCII downloader (the ~put command) to load S-records into the target board. The '--name' string must be a machine name that tip understands (for example, on some tip implementations it must be an entry from the initialization file for tip; this file is sometimes called '/etc/remote').

See your system documentation for information on how to create new entries in '/etc/remote'. (Some UNIX systems are distributed with at least one default entry with a name resembling 'hardwire'; if your system has one, you can edit it, or make a modified copy with a new name.) When you have a working '/etc/remote' entry *abugtarget*, you should be able to type 'tip *abugtarget*', and get the prompt '135ABUG>' from the board. Use the same *abugtarget* string with 'runtest --name'.

Motorola IDP board running the rom68k boot monitor

This is the same in functionality as the MVME board running the BUG boot monitor. Only the monitor commands and the addresses are different.

VxWorks (Motorola 68K or Intel 960)

Configure DejaGnu for either 'm68k-wrs-vxworks' (abbreviated 'vxworks68') or 'i960-wrs-vxworks' (abbreviated 'vxworks960'). Since both targets support IP addressing, specify the network address (for example, a host name from '/etc/hosts') with '--name'.

The default connect protocol is 'rlogin', but you can use any of '--connect rlogin', '--connect telnet', or '--connect rsh'.

Test scripts need no special code to load programs into these targets; since VxWorks supports NFS, all you must do is ensure test programs are on an exported filesystem.

When you compile for VxWorks, use the linker '-r' option to make the linker output relocatable—at least if you want to use library routines. Many standard C routines are included in VxWorks; often no additional libraries are needed. See your VxWorks system documentation for additional details.

4.7 The files DejaGnu reads

The `runtest` program used to invoke DejaGnu is a short shell script generated by `make` during the configuration process. Its main task is to read the main test framework driver, '`runtest.exp`'.

'`runtest.exp`', in turn, reads `expect` code from certain other files, in this order:

1. Each of the '`site.exp`' local definition files available. See Chapter 3 "Setting `runtest` defaults," page 15, for details.
2. '`lib/utils.exp`', a collection of utility procedures. See Section 4.4 "Library Procedures," page 26, for descriptions of these procedures.
3. '`framework.exp`', a file of subroutines meant for `runtest` itself rather than for general-purpose use in both `runtest` and test suites.
4. '`debugger.exp`', Don Libes' Tcl Debugger. (See *A Debugger for Tcl Applications* by Don Libes. This paper is distributed with `expect` in PostScript form as the file '`expect/tcl-debug.ps`'.)
5. An initialization file *`tool`*`_init.exp`. See Section 4.2 "Initialization module," page 22, for more discussion of init files.

4.8 The files DejaGnu writes

`runtest` always writes two kinds of output files: summary logs and detailed logs. The contents of both of these are determined by your tests.

For troubleshooting, a third kind of output file is useful: use '`--debug`' to request an output file showing details of what `expect` is doing internally.

4.8.1 Summary log

`runtest` always produces a summary output file '*tool*`.sum`'. This summary shows the names of all test files run; for each test file, one line of output from each `pass` command (showing status '`PASS`' or '`XPASS`') or `fail` command (status '`FAIL`' or '`XFAIL`'); trailing summary statistics that count passing and failing tests (expected and unexpected); and the full pathname and version number of the tool tested. (All possible outcomes, and all errors, are always reflected in the summary output file, regardless of whether or not you specify '`--all`'.)

If any of your tests use the procedures `unresolved`, `unsupported`, or `untested`, the summary output also tabulates the corresponding outcomes.

For example, after '`runtest --tool binutils`', look for a summary log in '`binutils.sum`'. Normally, `runtest` writes this file in your current working directory; use the '`--outdir`' option to select a different directory.

Here is a short sample summary log:

```
Test Run By rob on Mon May 25 21:40:57 PDT 1992
                === gdb tests ===
Running ./gdb.t00/echo.exp ...
PASS:   Echo test
Running ./gdb.all/help.exp ...
PASS:   help add-symbol-file
PASS:   help aliases
PASS:   help breakpoint "bre" abbreviation
FAIL:   help run "r" abbreviation
Running ./gdb.t10/crossload.exp ...
PASS:   m68k-elf (elf-big) explicit format; loaded
XFAIL:  mips-ecoff (ecoff-bigmips) "ptype v_signed_char" signed
C types
                === gdb Summary ===
# of expected passes 5
# of expected failures 1
# of unexpected failures 1
/usr/latest/bin/gdb version 4.6.5 -q
```

4.8.2 Detailed log

`runtest` also saves a detailed log file '*`tool`*`.log`', showing any output generated by tests as well as the summary output. For example, after '`runtest --tool binutils`', look for a detailed log in '`binutils.log`'. Normally, `runtest` writes this file in your current working directory; use the '`--outdir`' option to select a different directory.

Here is a brief example showing a detailed log for G++ tests:

```
Test Run By rob on Mon May 25 21:40:43 PDT 1992

                === g++ tests ===

--- Running ./g++.other/t01-1.exp ---
        PASS:   operate delete

--- Running ./g++.other/t01-2.exp ---
        FAIL:   i960 bug EOF
p0000646.C: In function `int  warn_return_1 ()':
p0000646.C:109: warning: control reaches end of non-void function
p0000646.C: In function `int  warn_return_arg (int)':
p0000646.C:117: warning: control reaches end of non-void function
p0000646.C: In function `int  warn_return_sum (int, int)':
p0000646.C:125: warning: control reaches end of non-void function
p0000646.C: In function `struct foo warn_return_foo ()':
p0000646.C:132: warning: control reaches end of non-void function

--- Running ./g++.other/t01-4.exp ---
        FAIL:   abort
900403_04.C:8: zero width for bit-field `foo'
--- Running ./g++.other/t01-3.exp ---
        FAIL:   segment violation
900519_12.C:9: parse error before `;'
900519_12.C:12: Segmentation violation
/usr/latest/bin/gcc: Internal compiler error: program cc1plus got
fatal signal

                === g++ Summary ===

# of expected passes 1
# of expected failures 3
/usr/ps/bin/g++ version cygnus-2.0.1
```

4.8.3 Logging `expect` internal actions

With the '`--debug`' option, you can request a log file showing the output from `expect` itself, running in debugging mode. This file ('`dbg.log`', in the directory where you start `runtest`) shows each pattern `expect` considers in analyzing test output.

This file reflects each `send` command, showing the string sent as input to the tool under test; and each `expect` command, showing each pattern it compares with the tool output.

The log messages for `expect` begin with a message of the form

```
expect: does {tool output} (spawn_id n) match pattern
{expected pattern}?
```

For every unsuccessful match, `expect` issues a '`no`' after this message; if other patterns are specified for the same `expect` command, they are reflected also, but without the first part of the message ('`expect. . .match pattern`').

When `expect` finds a match, the log for the successful match ends with '`yes`', followed by a record of the `expect` variables set to describe a successful match. Here is an excerpt from the debugging log for a GDB test:

```
send: sent {break gdbme.c:34\n} to spawn id 6
expect: does {} (spawn_id 6) match pattern {Breakpoint.*at.* file
 gdbme.c, line 34.*\(gdb\) $}? no
{.*\(gdb\) $}? no
expect: does {} (spawn_id 0) match pattern {<return>}? no
{\(y or n\) }? no
{buffer_full}? no
{virtual}? no
{memory}? no
{exhausted}? no
{Undefined}? no
{command}? no
break gdbme.c:34
Breakpoint 8 at 0x23d8: file gdbme.c, line 34.
(gdb) expect: does {break gdbme.c:34\r\nBreakpoint 8 at 0x23d8:
file gdbme.c, line 34.\r\n(gdb) } (spawn_id 6) match pattern
{Breakpoint.*at.* file gdbme.c, line 34.*\(gdb\) $}? yes
expect: set expect_out(0,start) {18}
expect: set expect_out(0,end) {71}
expect: set expect_out(0,string) {Breakpoint 8 at 0x23d8: file
gdbme.c, line 34.\r\n(gdb) }
expect: set expect_out(spawn_id) {6}
expect: set expect_out(buffer) {break gdbme.c:34\r\nBreakpoint 8
at 0x23d8: file gdbme.c, line 34.\r\n(gdb) }
        PASS:   70      0       breakpoint line number in file
```

This example exhibits three properties of `expect` and DejaGnu that might be surprising at first glance:

- Empty output for the first attempted match. The first set of attempted matches shown ran against the output '`{}`'—that is, no output. `expect` begins attempting to match the patterns supplied immediately; often, the first pass is against incomplete output (or completely before all output, as in this case).
- Interspersed tool output. The beginning of the log entry for the second attempted match may be hard to spot: this is because the prompt '`(gdb) `' appears on the same line, just before the '`expect:`' that marks the beginning of the log entry.

- Fail-safe patterns. Many of the patterns tested are fail-safe patterns provided by GDB testing utilities, to reduce possible indeterminacy. It is useful to anticipate potential variations caused by extreme system conditions (GDB might issue the message '`virtual memory exhausted`' in rare circumstances), or by changes in the tested program ('`Undefined command`' is the likeliest outcome if the name of a tested command changes).

  The pattern '`{<return>}`' is a particularly interesting fail-safe to notice; it checks for an unexpected RET prompt. This may happen, for example, if the tested tool can filter output through a pager.

  These fail-safe patterns (like the debugging log itself) are primarily useful while developing test scripts. Use the `error` procedure to make the actions for fail-safe patterns produce messages starting with '`ERROR`' on the `runtest` standard output, and in the detailed log file.

5 How To Write a Test Case

5.1 Writing a test case

The easiest way to prepare a new test case is to base it on an existing one for a similar situation. There are two major categories of tests: batch or interactive. Batch oriented tests are usually easier to write.

The GCC tests are a good example of batch oriented tests. All GCC tests consist primarily of a call to a single common procedure, since all the tests either have no output, or only have a few warning messages when successfully compiled. Any non-warning output is a test failure. All the C code needed is kept in the test directory. The test driver, written in `expect`, need only get a listing of all the C files in the directory, and compile them all using a generic procedure. This procedure and a few others supporting for these tests are kept in the library module '`lib/c-torture.exp`' in the GCC test suite. Most tests of this kind use very few `expect` features, and are coded almost purely in Tcl.

Writing the complete suite of C tests, then, consisted of these steps:

1. Copying all the C code into the test directory. These tests were based on the C-torture test created by Torbjorn Granlund (on behalf of the Free Software Foundation) for GCC development.
2. Writing (and debugging) the generic `expect` procedures for compilation.
3. Writing the simple test driver: its main task is to search the directory (using the Tcl procedure `glob` for filename expansion with wildcards) and call a Tcl procedure with each filename. It also checks for a few errors from the testing procedure.

Testing interactive programs is intrinsically more complex. Tests for most interactive programs require some trial and error before they are complete.

However, some interactive programs can be tested in a simple fashion reminiscent of batch tests. For example, prior to the creation of DejaGnu, the GDB distribution already included a wide-ranging testing procedure. This procedure was very robust, and had already undergone much more debugging and error checking than many recent DejaGnu test cases. Accordingly, the best approach was simply to encapsulate the existing GDB tests, for reporting purposes. Thereafter, new GDB tests built up a family of `expect` procedures specialized for GDB testing.

'`gdb.t10/crossload.exp`' is a good example of an interactive test.

5.2 Debugging a test case

These are the kinds of debugging information available from DejaGnu:

1. Output controlled by test scripts themselves, explicitly allowed for by the test author. This kind of debugging output appears in the detailed output recorded in the '*tool*.log' file. To do the same for new tests, use the `verbose` procedure (which in turn uses the variable also called `verbose`) to control how much output to generate. This will make it easier for other people running the test to debug it if necessary. Whenever possible, if '$verbose' is `0`, there should be no output other than the output from `pass`, `fail`, `error`, and `warning`. Then, to whatever extent is appropriate for the particular test, allow successively higher values of '$verbose' to generate more information. Be kind to other programmers who use your tests: provide for a lot of debugging information.
2. Output from the internal debugging functions of Tcl and `expect`. There is a command line options for each; both forms of debugging output are recorded in the file `dbg.log` in the current directory.

   Use '--debug' for information from the `expect` level; it generates displays of the `expect` attempts to match the tool output with the patterns specified (see Section 4.8.3 "Debug Log," page 36). This output can be very helpful while developing test scripts, since it shows precisely the characters received. Iterating between the latest attempt at a new test script and the corresponding 'dbg.log' can allow you to create the final patterns by "cut and paste". This is sometimes the best way to write a test case.

   Use '--strace' to see more detail at the Tcl level; this shows how Tcl procedure definitions expand, as they execute. The associated number controls the depth of definitions expanded; see the discussion of '--strace' in Chapter 2 "Running the Tests," page 9.
3. Finally, if the value of 'verbose' is 3 or greater, `runtest` turns on the `expect` command `log_user`. This command prints all `expect` actions to the `expect` standard output, to the detailed log file, and (if '--debug' is on) to 'dbg.log'.

5.3 Adding a test case to a test suite

There are two slightly different ways to add a test case. One is to add the test case to an existing directory. The other is to create a new directory to hold your test. The existing test directories represent several styles of testing, all of which are slightly different; examine the directories for the tool of interest to see which (if any) is most suitable.

Adding a GCC test can be very simple: just add the C code to any directory beginning with '`gcc.`' and it runs on the next '`runtest --tool gcc`'.

To add a test to GDB, first add any source code you will need to the test directory. Then you can either create a new `expect` file, or add your test to an existing one (any file with a '`.exp`' suffix). Creating a new '`.exp`' file is probably a better idea if the test is significantly different from existing tests. Adding it as a separate file also makes upgrading easier. If the C code has to be already compiled before the test will run, then you'll have to add it to the '`Makefile.in`' file for that test directory, then run `configure` and `make`.

Adding a test by creating a new directory is very similar:

1. Create the new directory. All subdirectory names begin with the name of the tool to test; e.g. G++ tests might be in a directory called '`g++.other`'. There can be multiple test directories that start with the same tool name (such as '`g++`').
2. Add the new directory name to the '`configdirs`' definition in the '`configure.in`' file for the test suite directory. This way when `make` and `configure` next run, they include the new directory.
3. Add the new test case to the directory, as above.
4. To add support in the new directory for configure and make, you must also create a `Makefile.in` and a `configure.in`. See section "What Configure Does" in *Cygnus Configure*.

5.4 Hints on writing a test case

There may be useful existing procedures already written for your test in the '`lib`' directory of the DejaGnu distribution. See Section 4.4 "Library Procedures," page 26.

It is safest to write patterns that match *all* the output generated by the tested program; this is called *closure*. If a pattern does not match the entire output, any output that remains will be examined by the *next* `expect` command. In this situation, the precise boundary that determines which `expect` command sees what is very sensitive to timing between the `expect` task and the task running the tested tool.

As a result, the test may sometimes appear to work, but is likely to have unpredictable results. (This problem is particularly likely for interactive tools, but can also affect batch tools—especially for tests that take a long time to finish.) The best way to ensure closure is to use the '-re' option for the `expect` command to write the pattern as a full regular expressions; then you can match the end of output using a '$'. It is also a good idea to write patterns that match all available output by using '.*\' after the text of interest; this will also match any intervening blank lines. Sometimes an alternative is to match end of line using '\r' or '\n', but this is usually too dependent on terminal settings.

Always escape punctuation, such as '(' or '"', in your patterns; for example, write '\('. If you forget to escape punctuation, you will usually see an error message like 'extra characters after close-quote'.

If you have trouble understanding why a pattern does not match the program output, try using the '--debug' option to `runtest`, and examine the debug log carefully. See Section 4.8.3 "Debug Log," page 36.

Be careful not to neglect output generated by setup rather than by the interesting parts of a test case. For example, while testing GDB, I issue a send 'set height 0\n' command. The purpose is simply to make sure GDB never calls a paging program. The 'set height' command in GDB does not generate any output; but running *any* command makes GDB issue a new '(gdb) ' prompt. If there were no `expect` command to match this prompt, the output '(gdb) ' begins the text seen by the next `expect` command—which might make *that* pattern fail to match.

To preserve basic sanity, I also recommended that no test ever pass if there was any kind of problem in the test case. To take an extreme case, tests that pass even when the tool will not spawn are misleading. Ideally, a test in this sort of situation should not fail either. Instead, print an error message by calling one of the DejaGnu procedures `error` or `warning`.

5.5 Special variables used by test cases

Your test cases can use these variables, with conventional meanings (as well as the variables saved in 'site.exp' see Chapter 3 "Setting runtest defaults," page 15):

These variables are available to all test cases.

`prms_id`
: The tracking system (e.g. GNATS) number identifying a corresponding bugreport. ('0' if you do not specify it in the test script.)

`bug_id`
: An optional bug id; may reflect a bug identification from another organization. ('0' if you do not specify it.)

`subdir` The subdirectory for the current test case.

These variables should never be changed. They appear in most tests.

`expect_out(buffer)`
: The output from the last command. This is an internal variable set by `expect`.

`exec_output`
: This is the output from a *tool*`_load` command. This only applies to tools like GCC and GAS which produce an object file that must in turn be executed to complete a test.

`comp_output`
: This is the output from a *tool*`_start` command. This is conventionally used for batch oriented programs, like GCC and GAS, that may produce interesting output (warnings, errors) without further interaction.

6 New Tools, Targets, or Hosts

The most common ways to extend the DejaGnu framework are: adding a suite of tests for a new tool to be tested; adding support for testing on a new target; and porting `runtest` to a new host.

6.1 Writing tests for a new tool

In general, the best way to learn how to write (code or even prose) is to read something similar. This principle applies to test cases and to test suites. Unfortunately, well-established test suites have a way of developing their own conventions: as test writers become more experienced with DejaGnu and with Tcl, they accumulate more utilities, and take advantage of more and more features of `expect` and Tcl in general.

Inspecting such established test suites may make the prospect of creating an entirely new test suite appear overwhelming. Nevertheless, it is quite straightforward to get a new test suite going.

There is one test suite that is guaranteed not to grow more elaborate over time: both it and the tool it tests were created expressly to illustrate what it takes to get started with DejaGnu. The '`example/`' directory of the DejaGnu distribution contains both an interactive tool called `calc`, and a test suite for it. Reading this test suite, and experimenting with it, is a good way to supplement the information in this section. (Thanks to Robert Lupton for creating `calc` and its test suite—and also the first version of this section of the manual!)

To help orient you further in this task, here is an outline of the steps to begin building a test suite for a program *`example`*.

1. Create or select a directory to contain your new collection of tests. Change to that directory (shown here as `testsuite`):

   ```
   eg$ cd testsuite/
   ```

2. Create a '`configure.in`' file in this directory, to control configuration-dependent choices for your tests. So far as DejaGnu is concerned, the important thing is to set a value for the variable `target_abbrev`; this value is the link to the init file you will write soon. (For simplicity, we assume the environment is Unix, and use '`unix`' as the value.)

   What else is needed in '`configure.in`' depends on the requirements of your tool, your intended test environments, and which `configure` system you use. This example is a minimal `configure.in` for use with Cygnus Configure. (For an alternative based on the FSF `autoconf` system, see the `calc` example distributed with DejaGnu.) Replace *`example`* with the name of your program:

```
# This file is a shell script fragment
# for use with Cygnus configure.

srctrigger="example.0"
srcname="The DejaGnu example tests"

# per-host:

# per-target:

# everything defaults to unix for a target
target_abbrev=unix

# post-target:
```

3. Create 'Makefile.in', the source file used by `configure` to build your 'Makefile'. Its leading section should as usual contain the values that `configure` may override:

```
srcdir = .
prefix = /usr/local

exec_prefix = $(prefix)
bindir = $(exec_prefix)/bin
libdir = $(exec_prefix)/lib
tooldir = $(libdir)/$(target_alias)

datadir = $(exec_prefix)/lib/dejagnu

RUNTEST = runtest
RUNTESTFLAGS =
FLAGS_TO_PASS =

#### host, target, site specific Makefile frags come in here.
```

This should be followed by the standard targets at your site. To begin with, they need not do anything—for example, these definitions will do:

```
all:

info:

install-info:

install:
uninstall:

clean:
        -rm -f *~ core *.info*
```

It is also a good idea to make sure your 'Makefile' can rebuild itself if 'Makefile.in' changes, with a target like this (which works for either Cygnus or FSF Configure):

```
Makefile : $(srcdir)/Makefile.in $(host_makefile_frag) \
             $(target_makefile_frag)
        $(SHELL) ./config.status
```

You also need to include two targets important to DejaGnu: `check`, to run the tests, and `site.exp`, to set up the Tcl copies of configuration-dependent values. The `check` target must run '`runtest --tool` *example*':

```
check: site.exp all
        $(RUNTEST) $(RUNTESTFLAGS) $(FLAGS_TO_PASS) \
            --tool example --srcdir $(srcdir)
```

The `site.exp` target should usually set up (among other things!) a Tcl variable for the name of your program:

```
site.exp: ./config.status Makefile
        @echo "Making a new config file..."
        -@rm -f ./tmp?
        @touch site.exp

        -@mv site.exp site.bak
        @echo "## these variables are automatically\
 generated by make ##" > ./tmp0
        @echo "# Do not edit here. If you wish to\
 override these values" >> ./tmp0
        @echo "# add them to the last section" >> ./tmp0
        @echo "set host_os ${host_os}" >> ./tmp0
        @echo "set host_alias ${host_alias}" >> ./tmp0
        @echo "set host_cpu ${host_cpu}" >> ./tmp0
        @echo "set host_vendor ${host_vendor}" >> ./tmp0
        @echo "set target_os ${target_os}" >> ./tmp0
        @echo "set target_alias ${target_alias}" >> ./tmp0
        @echo "set target_cpu ${target_cpu}" >> ./tmp0
        @echo "set target_vendor ${target_vendor}" >> ./tmp0
        @echo "set host_triplet ${host_canonical}" >> ./tmp0
        @echo "set target_triplet ${target_canonical}">>./tmp0
        @echo "set tool binutils" >> ./tmp0
        @echo "set srcdir ${srcdir}" >> ./tmp0
        @echo "set objdir `pwd`" >> ./tmp0
        @echo "set examplename example" >> ./tmp0
        @echo "## All variables above are generated by\
 configure. Do Not Edit ##" >> ./tmp0
                @cat ./tmp0 > site.exp
        @sed < site.bak \
               -e '1,/^## All variables above are.*##/ d' \
               >> site.exp
        -@rm -f ./tmp?
```

4. Create a directory (in '`testsuite/`') called '`config/`':

   ```
   eg$ mkdir config
   ```

5. Make an init file in this directory; its name must start with the `target_abbrev` value, so call it '`config/unix.exp`'. This is the file that contains the target-dependent procedures; fortunately, most of them do not have to do very much in order for `runtest` to run.

   If *`example`* is not interactive, you can get away with this minimal '`unix.exp`' to begin with:

   ```
   proc foo_exit {} {}
   proc foo_version {} {}
   ```

   If *`example`* is interactive, however, you might as well define a start routine *and invoke it* by using an init file like this:

```
proc foo_exit {} {}
proc foo_version {} {}

proc foo_start {} {
  global examplename
  spawn $examplename
  expect {
    -re "" {}
  }
}
foo_start
```

6. Create a directory whose name begins with your tool's name, to contain tests:

   eg$ mkdir *example*.0

7. Create a sample test file in '*example*.0'. Its name must end with '.exp'; you can use 'first-try.exp' To begin with, just write there a line of Tcl code to issue a message:

```
send_user "Testing: one, two...\n"
```

8. Back in the 'testsuite/' (top level) directory, run

   eg$ configure

   (You may have to specify more of a path, if a suitable configure is not available in your execution path.)

9. You are now ready to triumphantly type 'make check' or 'runtest --tool *example*'. You should see something like this:

```
Test Run By rhl on Fri Jan 29 16:25:44 EST 1993

                === example tests ===

Running ./example.0/first-try.exp ...
Testing: one, two...

                === example Summary ===
```

   There is no output in the summary, because so far the example does not call any of the procedures that establish a test outcome.

10. Begin writing some real tests. For an interactive tool, you should probably write a real exit routine in fairly short order; in any case, you should also write a real version routine soon.

6.2 Adding a target

DejaGnu has some additional requirements for target support, beyond the general-purpose provisions of Cygnus `configure`. `runtest` must actively communicate with the target, rather than simply generating or managing code for the target architecture. Therefore, each tool requires an initialization module for each target. For new targets, you must supply a few Tcl procedures to adapt DejaGnu to the target. This permits DejaGnu itself to remain target independent. See Section 4.2 "Initialization module," page 22, for a discussion of the naming conventions that enable DejaGnu to locate and use init files.

Usually the best way to write a new initialization module is to edit an existing initialization module; some trial and error will be required. If necessary, you can use the '`--debug`' option to see what is really going on.

When you code an initialization module, be generous in printing information controlled by the `verbose` procedure (see Section 4.3 "DejaGnu Builtins," page 23).

Most of the work is in getting the communications right. Communications code (for several situations involving IP networks or serial lines) is available in a DejaGnu library file, '`lib/remote.exp`'. See Section 4.4 "Library Procedures," page 26.

If you suspect a communication problem, try running the connection interactively from `expect`. (There are three ways of running `expect` as an interactive interpreter. You can run `expect` with no arguments, and control it completely interactively; or you can use '`expect -i`' together with other command-line options and arguments; or you can run the command `interpreter` from any `expect` procedure. Use `return` to get back to the calling procedure (if any), or `return -tcl` to make the calling procedure itself return to its caller; use `exit` or end-of-file to leave `expect` altogether.) Run the program whose name is recorded in '`$connectmode`', with the arguments in '`$targetname`', to establish a connection. You should at least be able to get a prompt from any target that is physically connected.

6.3 Porting to a new host

The task of porting DejaGnu is basically that of porting Tcl and `expect`. Tcl and `expect`, as distributed with DejaGnu, both use `autoconf`; they should port automatically to most Unix systems.

Once Tcl and `expect` are ported, DejaGnu should run. Most system dependencies are taken care of by using `expect` as the main command shell.

Appendix A Expect: Programmed Dialogue

This appendix presents the same information available in the Expect Version 5 reference-manual entry ("man page"); we include it in the DejaGnu manual for convenient reference. The author of Expect, and of this reference, is Don Libes, of the National Institute of Standards and Technology (NIST).

A.1 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`. You can also use Expect directly in C or C++ programs (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 several 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" or "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`, and so on.

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.

A.2 Usage

Expect reads a file for a list of commands to execute. You can also arrange to call Expect implicitly on systems which support the '`#!`' notation by marking the script executable, and making the first line in your script:

```
#!/usr/local/bin/expect -f
```

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

Here is a summary of the command line options you can use with Expect:

```
expect [ -dDinN ] [ -c cmds ] [[ -f ] cmdfile ] [ args ]
```

'`--`' 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 option-like interpretation by Expect. For example, the following will leave the original arguments (including the script name) in the variable *argv*.

```
#!/usr/local/bin/expect --
```

Note that the usual `getopt` and `execve` conventions must be observed when adding arguments to the '`#!`' line.

Expect understands these command line options:

`-c` *cmds*
: Execute the command *cmds* 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. You can execute multiple commands with a single '`-c`' by separating the commands with semicolons. Commands are executed in the order they appear. (If you use Expectk, specify this option as '`-command`'.)

`-d`
: Enables some debugging output, which primarily reports internal activity of commands such as `expect` and `interact`. This option 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, specify this option as '`-diag`'.)

`-D` *`lev`*
: Enables an interactive debugger. The option argument *`lev`* is an integer value. The debugger takes control before the next Tcl procedure if *`lev`* is non-zero or if a `C-C` is pressed (or a breakpoint is hit, or other appropriate debugger command appears in the script). See Section A.9 "Expect Bibliography," page 84, for more information on the debugger. (When using Expectk, specify this option as '`-Debug`'.)

`-f` *`cmdfile`*
: Read commands from *`cmdfile`*. The option itself is optional as it is only useful with using the '`#!`' notation in a script(see above), so that other arguments may be supplied on the command line. (When using Expectk, specify this option as '`-file`'.)

`-b`
: By default, Expect reads the command file into memory in its entirety before executing it. It is occasionally desirable to read files one line at a time. For example, stdin is read this way. In order to force Expect to handle arbitrary files this way, use the '`-b`' option. (When using Expectk, specify this option as '`-buffer`'.)

  If you supply the string '`-`' as a filename, Expect reads standard input instead. (Use '`./-`' to read from a file actually named '`-`'.)

`-i`
: 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 `interpreter` (below) for more information. Expect assumes '`-i`' if you use neither a command file nor '`-c`'. (When using Expectk, specify this option as '`-interactive`'.)

`-N`
`-n`
: The file '`$exp_library/expect.rc`' is sourced automatically if present, unless you use the '`-N`' option. (When using Expectk, specify this option as '`-NORC`'.) Immediately after this, the file '`~/.expect.rc`' is sourced

automatically, unless you use the '-n' option. (When using Expectk, specify this option as '-norc'.) Both of these are sourced after executing any '-c' options.

args 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"
```

A.3 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 Appendix B "Tcl," page 87). 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. Then read the examples at the rear of this man page. (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` *sw*] [`-i` *spawn_id*]

Closes the connection to the current process. Most interactive programs detect EOF on their standard input, and exit; thus `close` usually suffices to kill the process as well. The '-i' option 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' option determines whether the spawn id will be closed (when *sw* is nonzero) in any new spawned processes, or if you use `overlay` to replace the current process. To leave a spawn id open, use the value `0` for *sw*. The default is to force the spawn id closed in any new processes.

The '-slave' option closes the slave associated with the spawn id. (See 'open -pty'.) If you leave the slave open, Expect closes it automatically when the connection is closed.

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` below for more information.

`debug` [[`-now`] *sw*]

Controls a Tcl debugger that allows you to step through statements, set breakpoints, and so on.

When you run `debug` with no arguments, it returns `1` if the debugger is not running, or `0` otherwise.

To start the debugger, specify a value of `1` for the *sw* argument. To stop the debugger, specify a value of `0` for *sw*. Use the option '-now' with a *sw* value of `1` to start the debugger immediately (i.e., in the middle of the `debug` command itself). Otherwise, the debugger starts with the next Tcl statement.

The `debug` command does not change any traps. Compare this to starting Expect with the '-D' option (see Section A.2 "Usage," page 52).

See Section A.9 "Expect Bibliography," page 84, for more information on the debugger.

`exp_internal` [`-f` *file*] *value*

Causes further commands to send debugging information internal to Expect to stderr if *value* is non-zero. This output is disabled if *value* is 0. The debugging information includes every character received, and every attempt made to match the current output against the patterns.

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

`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 description for the `stty` command, which demonstrates how to turn off password echoing.)

```
send_user "password?\ "
expect_user -re "(.*)\n"
for {} 1 {} {
      if [fork]!=0 {exec 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` [`-onexit` [*`eh`*]] [`-noexit`] [*`status`*]

Exit Expect, or prepare to do so.

Use the '`-onexit` *`eh`*' option to set up a procedure named *`eh`* as an exit handler. Use '`exit -onexit`' (without an *`eh`* argument) to return the name of the current exit handler.

Use the '`-noexit`' option to prepare Expect to exit, but stop short of actually returning control to the operating system. The user-defined exit handler runs, as well as Expect's own internal handlers. Do not execute any further Expect commands. 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 the Expect `exit` is called again (however this might occur), the handlers do not re-run.

Upon exiting, all connections to spawned processes are closed. Closure will be detected as an EOF by spawned processes. `exit` takes no other actions beyond what the normal system subroutine `_exit` 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.) 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_pid` [`-i` *spawn_id*]
: Returns the process id corresponding to the currently spawned process. If you use the '`-i`' option, the result is the process id for the *spawn_id* you specify.

`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 *bodyn* 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. 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 two ways. By default, patterns are specified as with Tcl's `string match` command. (Such patterns are also similar to C-shell filename expansion, usually referred to as "glob" patterns).

Use the '`-gl`' option to distinguish patterns that might otherwise match `expect` options. You should protect any pattern beginning with a '-' by prefacing it with '`-gl`', since 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 {
  connected             break
  busy                  {
                          print busy\n
                          continue
                        }
  failed                abort
  "invalid password"    abort
  timeout               abort
}
```

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.

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

```
expect {
  connected                         break
  busy                              {
                                      print busy\n
                                      continue
                                    }
  -re "failed|invalid password" abort
  timeout                           abort
}
```

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.

The '-nocase' option 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_buffer`, the corresponding body is executed if *match_max* bytes have been received and no other patterns have matched.

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 array element '`expect_out(buffer)`'. Up to 9 regexp substring matches are saved in the array elements '`expect_out(1,string)`' through '`expect_out(9,string)`'.

If you use the option '-indices' before a pattern, `expect` saves the starting and ending indices of the matched strings. For each substring, '`expect_out(x,start)`' holds the starting index; the ending index is stored in '`expect_out(x,end)`'. These indices are in a form suitable for `lrange`; *x* corresponds to the substring position in the pattern. 0 refers to the entire pattern itself.

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. For 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 '`*`' (or '`-re ".*"`' flushes the output buffer without reading any more output from the process.

Normally, the matched output is discarded from Expect's internal buffers. You can prevent this by prefixing a pattern with the '`-notransfer`' option. This option is especially useful in experimenting (and, for convenience in that context, you can abbreviate it to '`-n`').

By default, patterns are matched against output from the current process; however, the '`-i`' option 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 shitespace-separated list of spawn ids, or a variable containing such a list.

For example, the following `expect` command waits for '`connected`' from the current process, or '`busy`', '`failed`' or '`invalid password`' from the *spawn_id* named by '`$proc2`'.

```
expect {
  connected                         break
  -i $proc2 busy                    {
                                      print busy\n
                                      continue
                                    }
  -re "failed|invalid password" abort
  timeout                           abort
}
```

The global variable `any_spawn_id` may be used to match patterns to any spawn ids that are named with all other '`-i`' options in t he current `expect` command. The spawn id from a '`-i`' option 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`.

Preface a pattern with the option '`-iwrite`' to set `expect_out(spawn_id)` to the spawn id which matched the pattern (or `eof`, or `full_buffer`). Prefacing a pattern with the option '`-iread`' causes any spawn id lists containing variables in any arguments of the current command to be reread when that pattern is matched. 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.

Prefacing a pattern with the option '`-timestamp`' updates two elements of `expect_out`: `expect_out(seconds)` and `expect_out(seconds_total)`. The `seconds` element is the number of seconds that have elapsed waiting for the current match. The `seconds_total` element is the number of seconds that have elapsed since the current `expect` command began running.

Actions such as `break` and `continue` cause control structures (i.e., `for`, `proc`) to behave in the usual way. The command '`exp_continue`' allows `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 `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
}
```

'exp_continue' resets the timeout timer.

`expect_after` [*expect-args*]
Takes the same arguments as `expect`; however, it returns immediately. Pattern-action pairs from the most recent `expect_after` 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` and `expect_after` can match, the `expect` pattern is used.

Unless overridden by a '`-i`' option, `expect_after` patterns match against the `spawn_id` defined at the time that the `expect_after` command was executed (not when its pattern is matched).

`expect_background` [*`expect-args`*]
: takes most of the same arguments as `expect`. However, it returns immediately. Patterns are tested whenever new input arrives. *Note: this only works in the Tk environment.* The patterns `timeout` and `default` are not meaningful to `expect_background`. `expect_background` accepts indirect spawn id specifications; however, they cannot be updated.

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. To cancel the effect of an `expect_background` for a particular spawn id, declare another `expect_background` with the same spawn id. Declaring `expect_background` with no pattern makes the corresponding spawn id stop matching any 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` 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.

Unless overridden by a '`-i`' option, `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).

`expect_user` [*expect-args*]

Like `expect`, but 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).

`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 up to 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_version` 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`' option, Expect prints an error and exits if the version is out of date.

`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 `C-Z` is pressed, Expect is suspended. (The `-reset` option restores the terminal modes.) When `C-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-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 0}
  \001    {send_user "you typed a control-A\n";
            send "\001"
          }
  $       {send_user "The date is [exec date]."}
  \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 string 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.) Use the '`-ex`' option to distinguish patterns that might otherwise match `interact` options. Protect any pattern beginning with a '-' by prefacing it with '`-ex`'. (All strings starting with '-' are reserved for future options.)

The '`-re`' option 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`' option 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`' option, in which case it is matched if end of file 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.

Preface a pattern with the option '`-iwrite`', to set `expect_out(spawn_id)` to the spawn id which matches the pattern (or eof). Prefacing a pattern with the option '`-iread`' causes any spawn id lists containing variables in any arguments of the current command to be re-read when that pattern is matched. This provides a way of changing the I/O source while the command is in execution.

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 `C-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 `C-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 press `ESC` (the escape character), and then press `C-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 '`-reset`' option 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 '`-reset`' is if your action depends on running in cooked mode.

By default, actions that change the value of `spawn_id` will not affect the behavior of interact even if input or output sources were originally associated with `spawn_id`.

The '`-update`' option forces `spawn_id` to be reexamined after evaluation of an action. This could be used, for example, so that pressing a particular function key would switch to interacting with a different process. Place the '`-update`' option before the pattern of the action.

The '`-echo`' option 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`' option 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 -f -re "(.*)\r" return
  puts $log "[exec date]: dialed $interact_out(1,string)"
}

interact -nobuffer -f "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`' option 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`' option (for "user") makes `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, preface each spawn id list with a '`-output`' option. Specify input for a group of output spawn ids by prefacing a spawn id list with a '`-input`' option. (Both '`-input`' and '`-output`' may take lists in the same form as the '`-i`' option in the '`expect`' command.) All following options and strings (or patterns) apply to this input until another '`-input`' option 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 overrides `$spawn_id`. Additional '`-input`' options 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`' option appears with no '`-output`' option, characters from that process are discarded.

The '`-i`' option introduces a replacement for the current `spawn_id` when no other '`-input`' or '`-output`' options are used.

`interpreter`

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.

`log_file` [[-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.

  The '`-a`' option forces output to be logged that was suppressed by the `log_user` command.

  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. A simple way to always start with a fresh log file is to delete the log file before using the `log_file` command for the first time in a script. For example:

```
exec rm transcript
log_file transcript
```

`log_user` *`expression`*
: By default, the send/expect dialogue is logged to standard output (and a logfile if open). To disable logging to stdout, use the command '`log_user 0`'; re-enable it with '`log_user 1`'. Logging to a logfile is unaffected by this command.

`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`' option, the default size is set. (The initial default is 2000.) With the '`-i`' option, the size is set for the named spawn id, otherwise it is set for the current process.

`overlay` [-*n* *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 descriptors *n*.

  Spawn ids are mapped to file descriptors 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' option, the default parity value is set. (The initial default is 1, i.e., parity is not stripped.) With the '-i' option, the parity value is set for the named spawn id, otherwise it is set for the current process.

`remove_nulls` [-d] [-i *spawn_id*] [*sw*]
: Defines whether null characters are retained, or removed from the output of spawned processes before pattern matching (and before storing in either `expect_out` or `interact_out`). If *sw* is zero, nulls are removed; otherwise they are not removed. With no *sw* argument, the current setting is returned.

  With the '-d' option, the default value is set. (The initial default is `1`, i.e., nulls are removed.) With the '-i' option, the value is set for the spawn id *spawn_id*; otherwise it is set for the current process.

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

`send [-s] [-h] [-i spawn_id] [-raw] [ -0 [ n ]] [ -- ] string`

Sends *string* to the current process. Strings are interpreted following Tcl rules. For example, the command

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

sends the characters 'h' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd' 'RET' to the current process. (Tcl includes a command similar to `printf` (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`'.

Use the '`--`' option to force the next argument to be interpreted as a string rather than an option. Any string can be preceded by '`--`' whether or not it actually looks like an option. This provides a reliable mechanism to specify variable strings without being tripped up by those that accidentally look like options. (All strings starting with '`-`' are reserved for future options.)

The '`-i`' option 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 it the terminal was in cooked mode. The '`-raw`' option disables this translation.

The '`-0`' option sends ASCII NUL characters (zeros). By default, one NUL is sent. Use an integer following the '`-0`' to specify how many NULs to send.

The '`-s`' option 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`' option 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 effective average can be quite different from the specified average if the minimum and miximum 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.

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 hints on how to break in,
# there is no prompt for an external password.
# Wait for 5 seconds for exec to complete
spawn telnet very.secure.gov
exec 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 provided for compatibility between environments. Similar aliases are provided for other Expect's other send commands.

`send_error` *string*

Like `send`, except that the string is sent to standard error rather than the current process.

send_log *string*
: Like send, except that *string* is only sent to the log file (see log_file). *string* is ignored if no log file is open.

send_spawn *args*
: An alias for send. If you are use expectk or some other variant of Expect in the Tk environment, send is defined by Tk for an entirely different purpose. send_spawn is provided for compatibility between environments.

send_user *args*
: is like send, except that the arguments are sent to stdout rather than the current process.

spawn [*opts*] 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 descriptors.

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 *current 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.

tty_spawn_id is a global variable containing a descriptor for '/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, the result is 0. The variable element 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' option stops spawn from doing this.

The '-console' option 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` fails (e.g. when *`program`* does not 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`' option causes the next argument to be interpreted as a Tcl file descriptor (i.e., a result of `open`.) You can then use the spawn id as if it were a spawned process. (The file descriptor 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.

The '`-pty`' option 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 element `spawn_out(slave,fd)` is set to a file descriptor corresponding to the pty slave. You can close it using '`close -slave`'.

`strace` *`level`*

Causes following statements to be printed before being executed. (The Tcl `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
```

`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 cooked mode. The arguments '`echo`' and '`-echo`' put the terminal into echo and noecho mode respectively.

The following example illustrates how to use `stty` to temporarily disable echoing. This could be used in otherwise-automatic scripts to avoid embedding passwords in them. (For more discussion of this, see Section A.8 "Expect Hints," page 82.)

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

`system` *args*

Gives *args* to `sh` as input, just 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` [`-format` *string*] [`-seconds` *n*]

Returns a timestamp. With no arguments, the result is the number of seconds since the epoch.

The '`-format`' option introduces a string which specifies a format for the result according to the POSIX rules for the `strftime` C subroutine. In the format string, you can specify these substitutions:

`%a`	abbreviated weekday name (e.g., '`Sat`')
`%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 (1–31)
`%H`	hour (0–23)
`%I`	hour (1–12)
`%j`	day (1–366)
`%m`	month (1–12)
`%M`	minute (0–59)
`%p`	'`am`' or '`pm`'
`%S`	second (0–61)
`%U`	week (0–50, Sunday is first day of week)
`%w`	day (0–6)
`%W`	week (0–50, Monday is first day of week)
`%x`	date, as in: '`Wed Oct 6 1993`'
`%X`	time, as in: '`23:59:59`'
`%y`	year of century (0–99)
`%Y`	year, as in: '`1993`'
`%%`	a percent sign

Other '`%`' specifications are undefined. Other characters are copied as specified. The final string returned must not be more than 200 characters. Only the C locale is supported.

The '`-seconds`' option introduces a number of seconds since the epoch, as the time to format. Otherwise, the current time is used.

`trap` [[*command*] *signals*]

Causes the given *command* to be executed upon future receipt of any of the given signals. The command executes with 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 signal handlers are reset 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. `ONEXIT` (signal 0) is raised upon exit from Expect.

For example, '`trap {send_user "Ouch!"} SIGINT`' prints '`Ouch!`' each time the user presses `C-C`.

With no arguments (or with the option '`-number`'), `trap` returns the signal number of the trap command currently executing.

The '`-code`' option uses the return code of the command in place of whatever Tcl was about to return when the command originally started running.

The '`-interp`' option 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`' option causes the `trap` command to return the signal name of the trap command currently being executed.

The '`-max`' option causes the `trap` command to return the largest signal number that can be set.

By default, `SIGINT` (which can usually be generated by pressing `C-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`' option to start the debugger, the `SIGINT` handler is redefined to start the interactive debugger, with the following trap:

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

You can set the environment variable `EXPECT_DEBUG_INIT` to change the debugger trap to another `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 overrides 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 information.

`wait [-i` *spawn_id*`]`

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

`wait` returns a list of four integers. The first integer is the process id 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, `0` otherwise. When the third integer is `0`, the fourth integer is the status returned by the spawned process. When the third integer is `-1`, the fourth integer is the value of `errno` set by the operating system. The global variable `errorCode` is also set.

The '`-i`' option declares the process to wait corresponding to the named spawn id (*not* the process id). You can wait for a signal in any process by using the value stored in the global variable `any_spawn_id`.

A.4 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
```

A.5 Examples

It may not be apparent how to put everything together that is described here. 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 examples in the Expect papers (see Section A.9 "Expect Bibliography," page 84) are also useful. While some papers use syntax corresponding to earlier versions of Expect, the accompanying rationales are stil valid, and go into a lot more detail than this document.

A.6 Caveats

Extensions may collide with Expect command names. For example, Tk defines `send` for an entirely different purpose. For this reason, most of the Expect commands are also available as 'exp_*cmd*'. Commands and variables beginning with 'exp', 'inter', 'spawn', and 'timeout' do not have aliases. Use the prefixed command names for 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`' characters with non-whitespace characters between them. This seems sufficiently improbable; however, the argument '`-brace`' 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.

A.7 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.

Telnet (verified only under SunOS 4.1.2) hangs if `TERM` is not set. This is a problem under `cron` and `at`, 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
```

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
exec 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 requires 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";
exec sleep 1
expect {
  timeout {send "\r"; exp_continue}
  $prompt
}
```

A.8 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` does not exist, this code still has a good chance of functioning correctly.

```
set prompt "(%|#|\\$) $"          ;# default prompt
if [info exists env(EXPECT_PROMPT)] {
  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 '` `'. 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 time out—instead it will get an `eof` indication. Use that instead. Even better, use both. That way if that line is ever moved around, you will not 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 do not.) 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.

A.9 Expect Bibliography

See Appendix B "Tcl," page 87, for discussion of the Tcl language that underlies Expect.

See *libexpect(3)*, for discussion of a library of C functions implementing `expect` functionality.

A number of papers provide further reading:

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

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

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

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

Regression Testing and Conformance Testing Interactive Programs
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
Don Libes, *Software—Practice & Experience*, John Wiley & Sons, West Sussex, England, Vol. 23, No. 5, May, 1993.

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

A.10 Acknowledgements

Don Libes, of the National Institute of Standards and Technology (NIST), implemented Expect.

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 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.

Appendix B Tcl Overview

This appendix presents the same information available in the Tool Command Language (Tcl) reference-manual entries ("man pages"); we include it in the DejaGnu manual for convenient reference. The author of Tcl, and of the reference-manual entries where this information originally appeared, is John Ousterhout, of the University of California at Berkeley (`ouster@sprite.berkeley.edu`).

The following copyright terms apply to the documentation in this Appendix:

B.1 Summary of Tcl language syntax

The following rules define the syntax and semantics of the Tcl language:

1. A Tcl script is a string containing one or more commands. Semicolons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

2. Tcl evaluates commands in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a command procedure to carry out the command, then all of the words of the command are passed to the command procedure. The command procedure is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.
3. Words of a command are separated by whitespace (except for newlines, which are command separators).
4. If the first character of a word is double-quote (`"`) then the word is terminated by the next double-quote character. If semicolons, close brackets, or whitespace characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double-quotes are not retained as part of the word.
5. If the first character of a word is an open brace (`{`) then the word is terminated by the matching close brace (`}`). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semicolons, newlines, close brackets, or whitespace have any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.
6. If a word contains an open bracket (`[`) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (`]`). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.
7. If a word contains a dollar-sign (`$`) then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substition may take any of the following forms:

`$name` — *name* is the name of a scalar variable; the name is terminated by any character that isn't a letter, digit, or underscore.

`$name(index)` — *name* gives the name of an array variable and *index* gives the name of an element within that array. *name* must contain only letters, digits, and underscores. Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of *index*.

`${name}` — *name* is the name of a scalar variable. It may contain any characters whatsoever except for close braces.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

8. If a backslash (\) appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

`\a` — Audible alert (bell) (`0x7`).

`\b` — Backspace (`0x8`).

`\f` — Form feed (`0xc`).

`\n` — Newline (`0xa`).

`\r` — Carriage-return (`0xd`).

`\t` — Tab (`0x9`).

`\v` — Vertical tab (`0xb`).

`\NEWLINE WHITESPACE` — A single space character replaces the backslash, `NEWLINE`, and all `WHITESPACE` characters after the `NEWLINE`. This backslash sequence is unique in that it is replaced in a separate preprocessing pass before the command is actually parsed. This means that it will be replaced *even when it occurs between braces*, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

`\\` — Backslash (\).

`\ooo` The digits ooo (one, two, or three of them) give the octal value of the character.

`\xhh` The hexadecimal digits *hh* give the hexadecimal value of the character. Any number of digits may be present.

Backslash substitution is not performed on words enclosed in braces, except for `\NEWLINE` as described above.

9. If a hash character (`#`) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.
10. Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command.

    For example, if variable substition occurs then no further substitions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are perfomed before making the recursive call and no additional substitutions are performed on the result of the nested script.
11. Substitutions do not affect the word boundaries of a command. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

B.2 Writing expressions in Tcl

A Tcl expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression processor.

The command `expr` evaluates expressions with no further effect. See Section B.3.15 "Evaluate an expression," page 103. The conditional command `if`, and two of the looping commands (`for` and `while`) also evaluate Tcl expressions to test conditions.

For some examples of simple expressions, suppose the variable `a` has the value `3` and the variable `b` has the value `6`. Then the command on the left side of each of the lines below will produce the value on the right side of the line:

```
expr 3.1 + $a                     6.1
expr 2 + "$a.$b"                  5.6
expr 4*[llength "6 2"]            8
expr {{word one} < "word $a"}     0
```

B.2.1 Operands in Tcl expressions

Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in octal (if the first character of the operand is '0'), or in hexadecimal (if the first two characters of the operand are '0x'). If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler (except that the 'f', 'F', 'l', and 'L' suffixes will not be permitted in most installations). For example, all of the following are valid floating-point numbers: '2.1', '3.', '6e4', '7.91e+16'. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

1. As an numeric value, either integer or floating-point.
2. As a Tcl variable, using standard '$' notation. The variable's value will be used as the operand.
3. As a string enclosed in double-quotes ("). The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand.
4. As a string enclosed in '{}' braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.
5. As a Tcl command enclosed in '[]' brackets. The command will be executed and its result will be used as the operand.
6. As a mathematical function whose arguments have any of the above forms for operands, such as 'sin($x)'. See Section B.2.3 "Mathematical functions in Tcl," page 93, for a list of defined functions.

Where substitutions occur above (e.g. inside quoted strings), they are performed by the expression processor. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents.

B.2.2 Operators in Tcl expressions

The operators Tcl recognizes in expressions are listed below, grouped in decreasing order of precedence:

`- ~ !` Unary minus, bit-wise NOT, logical NOT. None of these operands may be applied to string operands, and bit-wise NOT may be applied only to integers.

`* / %` Multiply, divide, remainder. None of these operands may be applied to string operands, and remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the divisor.

`+ -` Add and subtract. Valid for any numeric operands.

`<< >>` Left and right shift. Valid for integer operands only.

`< > <= >=` Boolean less, greater, less than or equal, and greater than or equal. Each operator produces `1` if the condition is true, `0` otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

`== !=` Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

`&` Bit-wise AND. Valid for integer operands only.

`^` Bit-wise exclusive OR. Valid for integer operands only.

`|` Bit-wise OR. Valid for integer operands only.

`&&` Logical AND. Produces a `1` result if both operands are non-zero, `0` otherwise. Valid for numeric operands only (integers or floating-point).

`||` Logical OR. Produces a `0` result if both operands are zero, `1` otherwise. Valid for numeric operands only (integers or floating-point).

x `?` *y* `:` *z* If-then-else, as in C. If *x* evaluates to non-zero, then the result is the value of *y*. Otherwise the result is the value of *z*. The *x* operand must have a numeric value.

See the C manual for more details on the results produced by each operator. All of the binary operators group left-to-right within the same precedence level. For example, the command

```
expr 4*2 < 7
```

returns 0.

The &&, ||, and ?: operators have “lazy evaluation”, just as in C, which means that operands are not evaluated if they are not needed to determine the outcome. For example, in the command

```
expr {$v ? [a] : [b]}
```

only one of ‘[a]’ or ‘[b]’ will actually be evaluated, depending on the value of ‘$v’. Note, however, that this is only true if the entire expression is enclosed in braces; otherwise the Tcl parser will evaluate both ‘[a]’ and ‘[b]’ before invoking the `expr` command.

B.2.3 Mathematical functions in Tcl

Tcl supports the following mathematical functions in expressions:

```
acos     cos      hypot    sinh
asin     cosh     log      sqrt
atan     exp      log10    tan
atan2    floor    pow      tanh
ceil     fmod     sin
```

Each of these functions invokes the C math library function of the same name; see the manual entries for the library functions for details on what they do. Tcl also implements the following functions for conversion between integers and floating-point numbers:

`abs(`*arg*`)`
: Returns the absolute value of *arg*. *arg* may be either integer or floating-point, and the result is returned in the same form.

`double(`*arg*`)`
: If *arg* is a floating value, returns *arg*; otherwise converts *arg* to floating and returns the converted value.

`int(`*arg*`)`
: If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by truncation and returns the converted value.

`round(`*arg*`)`
: If *arg* is an integer value, returns *arg*, otherwise converts *arg* to integer by rounding and returns the converted value.

B.2.4 Types, overflow, and precision in Tcl

All internal computations involving integers are done with the C type `long`, and all internal computations involving floating-point are done with the C type `double`. When converting a string to floating-point, exponent overflow is detected and results in a Tcl error. For conversion to integer from string, detection of overflow depends on the behavior of some routines in the local C library, so it should be regarded as unreliable. In any case, integer overflow and underflow are generally not detected reliably for intermediate results. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

```
expr 5 / 4
```

returns `1`, while

```
expr 5 / 4.0
expr 5 / ( [string length "abcd"] + 0.0 )
```

both return `1.25`. Floating-point values are always returned with a '`.`' or an '`e`' so that they will not look like integer values. For example,

```
expr 20.0/5.0
```

returns '`4.0`', not '`4`'. The global variable `tcl_precision` determines the the number of significant digits that are retained when floating values are converted to strings (except that trailing zeroes are omitted). If `tcl_precision` is unset then 6 digits of precision are used. To retain all of the significant bits of an IEEE floating-point number set `tcl_precision` to `17`; if a value is converted to string with 17 digits of precision and then converted back to binary for some later calculation, the resulting binary value is guaranteed to be identical to the original one.

B.2.5 String operations in Tcl

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string using the C `sprintf` format specifier '`%d`' for integers and '`%g`' for floating-point values. For example, the commands

```
expr {"0x03" > "2"}
expr {"0y" < "0x12"}
```

both return `1`. The first comparison is done using integer comparison, and the second is done using string comparison after the second operand is converted to the string '`18`'.

B.3 Commands built into Tcl

Here are descriptions of all the commands built into the Tcl language itself.

B.3.1 Append to variable: `append`

`append` *`varname value`* `[` *`value`* `... ]`

Append all of the *`value`* arguments to the current value of variable *`varname`*. If *`varname`* doesn't exist, it is given a value equal to the concatenation of all the *`value`* arguments. This command provides an efficient way to build up long variables incrementally. For example, '`append a $b`' is much more efficient than '`set a $a$b`' if `$a` is long.

B.3.2 Manipulate array variables: `array`

`array` *`option arrayname`* `[` *`arg`* `... ]`

This command performs one of several operations on the variable given by *`arrayname`*. *`arrayname`* must be the name of an existing array variable. The *`option`* argument determines what action is carried out by the command. The options (which may be abbreviated) are:

`array names` *`arrayname`*
: Returns a list containing the names of all of the elements in the array. If there are no elements in the array then an empty string is returned.

`array size` *`arrayname`*
: Returns a decimal string giving the number of elements in the array.

`array startsearch` *`arrayname`*
: This command initializes an element-by-element search through the array given by *`arrayname`*, such that invocations of the '`array nextelement`' command will return the names of the individual elements in the array. When the search has been completed, the '`array donesearch`' command should be invoked. The return value is a search identifier that must

be used in '`array nextelement`' and '`array donesearch`' commands; it allows multiple searches to be underway simultaneously for the same array.

`array nextelement` *`arrayname srchid`*
: Returns the name of the next element in *`arrayname`*, or an empty string if all elements of *`arrayname`* have already been returned in this search. The *`srchid`* argument identifies the search, and must have been the return value of an '`array startsearch`' command.

  Warning: if elements are added to or deleted from the array, then all searches are automatically terminated just as if '`array donesearch`' had been invoked; this will cause '`array nextelement`' operations to fail for those searches.

`array anymore` *`arrayname srchid`*
: Returns `1` if there are any more elements left to be processed in an array search, `0` if all elements have already been returned. *`srchid`* indicates which search on *`arrayname`* to check, and must have been the return value from a previous invocation of '`array startsearch`'. This option is particularly useful if an array has an element with an empty name, since the return value from '`array nextelement`' won't indicate whether the search has been completed.

`array donesearch` *`arrayname srchid`*
: This command terminates an array search and destroys all the state associated with that search. *`srchid`* indicates which search on *`arrayname`* to destroy, and must have been the return value from a previous invocation of '`array startsearch`'. Returns an empty string.

B.3.3 Abort looping: `break`

`break`

This command may be invoked only inside the body of a looping command such as `for` or `foreach` or `while`. It returns a `TCL_BREAK` code to signal the innermost containing loop command to return immediately.

B.3.4 Obsolescent command: `case`

```
case str [ in] patlist body [ patlist body ...]
case str [ in] {patlist body [ patlist body ...]}
```

Warning: the `case` command is obsolete, and is supported only for backward compatibility. At some point in the future it may be removed entirely. You should use the `switch` command instead.

The `case` command matches *`str`* against each of the *`patlist`* arguments in order. Each *`patlist`* argument is a list of one or more patterns. If any of these patterns matches *`str`* then `case` evaluates the following *`body`* argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. Each *`patlist`* argument consists of a single pattern or list of patterns. Each pattern may contain any of the wild-cards described under '`string match`' (see Section B.3.53 "Manipulate strings: `string`," page 132). If a *`patlist`* argument is the word `default`, the corresponding *`body`* will be evaluated if no *`patlist`* matches *`str`*. If no *`patlist`* argument matches *`str`* and no default is given, then the `case` command returns an empty string.

Two syntaxes are provided for the *`patlist`* and *`body`* arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line `case` commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *`patlist`* arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

B.3.5 Trap exceptions: `catch`

```
catch script [ varname ]
```

The `catch` command may be used to prevent errors from aborting command interpretation. `catch` calls the Tcl interpreter recursively to execute *`script`*, and always returns a `TCL_OK` code, regardless of any errors that might occur while executing *`script`*. The return value from `catch` is a decimal string giving the code returned by the Tcl interpreter after executing *`script`*. This will be 0 (`TCL_OK`) if there were no errors in *`script`*; otherwise it will have a non-zero value corresponding to one of the exceptional return codes (see Section B.3.47 "Return from a procedure: `return`," page 127, for descriptions of exception code values).

If the *varname* argument is given, then it gives the name of a variable; `catch` will set the variable to the string returned from *script* (either a result or an error message).

B.3.6 Change working directory: `cd`

`cd [ dirname ]`

Change the current working directory to *dirname*, or to the home directory (as specified in the `HOME` environment variable) if *dirname* is not given. Returns an empty string.

If *dirname* starts with a tilde, then tilde-expansion is done: the first element of *dirname* is replaced with the location of the home directory for the given user. The substitution is carried out in the same way that it would be done by `csh`. If the tilde is followed immediately by a slash, then the `HOME` environment variable is substituted. Otherwise the characters between the tilde and the next slash are taken as a user name, which is looked up in the password file; the user's home directory is retrieved from the password file and substituted.

B.3.7 Close an open file: `close`

`close fileid`

Closes the file given by *fileid*. *fileid* must be the return value from a previous invocation of the `open` command; after this command, it should not be used anymore. If *fileid* refers to a command pipeline instead of a file, then `close` waits for the children to complete. The normal result of this command is an empty string, but errors are returned if there are problems in closing the file or waiting for children to complete.

B.3.8 Join lists together: `concat`

`concat [ arg ... ]`

This command treats each argument as a list and concatenates them into a single list. It also eliminates leading and trailing spaces in each *arg* and adds a single separator space between each *arg*. It permits any number of arguments. For example, the command

```
concat a b {c d e} {f {g h}}
```

will return as its result

```
a b c d e f {g h}
```

If no *arg* is supplied, the result is an empty string.

B.3.9 Skip to the next iteration: `continue`

`continue`

This command may be invoked only inside the body of a looping command such as `for` or `foreach` or `while`. It returns a `TCL_CONTINUE` code to signal the innermost containing loop command to skip the remainder of the loop's body but continue with the next iteration of the loop.

B.3.10 Check for end-of-file: `eof`

`eof` *`fileid`*

Returns 1 if an end-of-file condition has occurred on *`fileid`*, 0 otherwise. *`fileid`* must be the return value from a previous call to `open`, or it may be `stdin`, `stdout`, or `stderr` to refer to one of the standard I/O channels.

B.3.11 Generate an error message: `error`

`error` *`message`* [*`info`*] [*`code`*]

Returns a `TCL_ERROR` code, which causes command interpretation to be unwound. *`message`* is a string that is returned to the application to indicate what went wrong.

If the *`info`* argument is provided and is non-empty, it is used to initialize the global variable `errorInfo`. `errorInfo` is used to accumulate a stack trace of what was in progress when an error occurred; as nested commands unwind, the Tcl interpreter adds information to `errorInfo`. If the *`info`* argument is present, it is used to initialize `errorInfo` and the first increment of unwind information will not be added by the Tcl interpreter. In other words, the command containing the error command will not appear in `errorInfo`; in its place will be *`info`*. This feature is most useful in conjunction with the `catch` command. If a caught error cannot be handled successfully, *`info`* can be used to return a stack trace reflecting the original point of occurrence of the error:

```
catch { ... } errMsg
set savedInfo $errorInfo
 ...
error $errMsg $savedInfo
```

If the *`code`* argument is present, then its value is stored in the `errorCode` global variable. This variable is intended to hold a machine-readable description of the error in cases where such information is available; see Section B.4 "Variables built into Tcl," page 141, for information on the proper format for the variable. If the *`code`* argument is not present, then `errorCode` is automatically reset to '`NONE`' by the Tcl interpreter as part of processing the error generated by the command.

B.3.12 Evaluate a Tcl script: `eval`

`eval` *arg* `[` *arg* `... ]`

`eval` takes one or more arguments, which together comprise a Tcl script containing one or more commands. `eval` concatenates all its arguments in the same fashion as the `concat` command, passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it).

B.3.13 Invoke subprocesses: `exec`

`exec [` *opts* `]` *arg* `[` *arg* `... ]`

This command treats its arguments as the specification of one or more subprocesses to execute. The arguments take the form of a standard shell pipeline where each *arg* becomes one word of a command, and each distinct command becomes a subprocess.

If the initial arguments to `exec` start with '–' then they are treated as command-line options and are not part of the pipeline specification. The following options are currently supported:

`-keepnewline`
: Retains a trailing newline in the pipeline's output. Normally a trailing newline will be deleted.

`--`
: Marks the end of *opts*. The argument following this one will be treated as the first *arg* even if it starts with a '–'.

If an *arg* (or pair of *arg* strings) has one of the forms described below then it is used by `exec` to control the flow of input and output among the subprocess(es). Such arguments will not be passed to the subprocess(es). In forms such as '`<` *fname*' *fname* may either be in a separate argument from '`<`' or in the same argument with no intervening space (i.e. '`<`*fname*').

`|`
: Separates distinct commands in the pipeline. The standard output of the preceding command will be piped into the standard input of the next command.

`|&`
: Separates distinct commands in the pipeline. Both standard output and standard error of the preceding command will be piped into the standard input of the next command. This form of redirection overrides forms such as '`2>`' and '`>&`'.

`<` *fname*
: The file named by *fname* is opened and used as the standard input for the first command in the pipeline.

`<@` *fileid*
: *fileid* must be the identifier for an open file, such as the return value from a previous call to `open`. It is used as the standard input for the first command in the pipeline. *fileid* must have been opened for reading.

`<<` *value*
: *value* is passed to the first command as its standard input.

`>` *fname*
: Standard output from the last command is redirected to the file named *fname*, overwriting its previous contents.

`2>` *fname*
: Standard error from all commands in the pipeline is redirected to the file named *fname*, overwriting its previous contents.

`>&` *fname*
: Both standard output from the last command and standard error from all commands are redirected to the file named *fname*, overwriting its previous contents.

`>>` *fname*
: Standard output from the last command is redirected to the file named *fname*, appending to it rather than overwriting it.

`2>>` *fname*
: Standard error from all commands in the pipeline is redirected to the file named *fname*, appending to it rather than overwriting it.

`>>&` *fname*
: Both standard output from the last command and standard error from all commands are redirected to the file named *fname*, appending to it rather than overwriting it.

`>` *fileid*
: *fileid* must be the identifier for an open file, such as the return value from a previous call to `open`. Standard output from the last command is redirected to the corresponding file, which must have been opened for writing.

`2>` *fileid*
: *fileid* must be the identifier for an open file, such as the return value from a previous call to `open`. Standard error from all commands in the pipeline is redirected to the corresponding file. The file must have been opened for writing.

`>&` *fileid*
: *fileid* must be the identifier for an open file, such as the return value from a previous call to `open`. Both standard output from the last command and standard error from all commands are redirected to the corresponding file. The file must have been opened for writing.

If standard output has not been redirected then the `exec` command returns the standard output from the last command in the pipeline. If any of the commands in the pipeline exit abnormally or are killed or suspended, then `exec` will return an error and the error message will include the pipeline's output followed by error messages describing the abnormal terminations; the `errorCode` variable will contain additional information about the last abnormal termination encountered. If any of the commands writes to its standard error file and that standard error isn't redirected, then `exec` will return an error; the error message will include the pipeline's standard output, followed by messages about abnormal terminations (if any), followed by the standard error output.

If the last character of the result or error message is a newline then that character is normally deleted from the result or error message. This is consistent with other Tcl return values, which don't normally end with newlines. However, if '`-keepnewline`' is specified then the trailing newline is retained.

If standard input isn't redirected with '<' or '<<' or '`<@`' then the standard input for the first command in the pipeline is taken from the application's current standard input.

If the last *arg* is '`&`' then the pipeline will be executed in background. In this case the `exec` command will return a list whose elements are the process identifiers for all of the subprocesses in the pipeline. The standard output from the last command in the pipeline will go to the application's standard output if it hasn't been redirected, and error output from all of the commands in the pipeline will go to the application's standard error file unless redirected.

The first word in each command is taken as the command name; tilde-substitution is performed on it, and if the result contains no slashes then the directories in the `PATH` environment variable are searched for an executable by the given name. If the name contains a slash then it must refer to an executable reachable from the current directory. No wildcard ("glob") expansion or other shell-like substitutions are performed on the arguments to commands.

B.3.14 End the application: `exit`

`exit [` *returncode* `]`

Terminate the process, returning *returncode* to the system as the exit status. If *returncode* isn't specified then it defaults to 0.

B.3.15 Evaluate an expression: `expr`

`expr arg ?arg arg ...?`

Concatenates arg's (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value. The operators permitted in Tcl expressions are a subset of the operators permitted in C expressions, and they have the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

```
expr 8.2 + 6
```

evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons.

B.3.16 File names and attributes: `file`

`file` *`opt fname`* [*`arg`* `...`]

This command provides several operations on a file's name or attributes. *`fname`* is the name of a file; if it starts with a tilde, then tilde substitution is done before executing the command (as for `cd`; see Section B.3.6 "Change working directory: `cd`," page 98). *`opt`* indicates what to do with the file name. Any unique abbreviation for *`opt`* is acceptable. The options currently implemented are:

`file atime` *`fname`*
: Returns a decimal string giving the time at which file *`fname`* was last accessed. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its access time cannot be queried then an error is generated.

`file dirname` *`fname`*
: Returns all of the characters in *`fname`* up to but not including the last slash character. If there are no slashes in *`fname`* then returns '.'. If the last slash in *`fname`* is its first character, then returns '/'.

`file executable` *`fname`*
: Returns `1` if file *`fname`* is executable by the current user, `0` otherwise.

`file exists` *`fname`*
: Returns `1` if file *`fname`* exists and the current user has search privileges for the directories leading to it, `0` otherwise.

`file extension` *`fname`*
: Returns all of the characters in *`fname`* after and including the last dot in *`fname`*. If there is no dot in *`fname`* then returns the empty string.

`file isdirectory` *`fname`*
: Returns `1` if file *`fname`* is a directory, `0` otherwise.

`file isfile` *`fname`*
: Returns `1` if file *`fname`* is a regular file, `0` otherwise.

`file lstat` *`fname varname`*
: Same as '`file stat`' (below), except using the `lstat` kernel call instead of `stat`. This means that if *`fname`* refers to a symbolic link the information returned in *`varname`* is for the link rather than the file it refers to. On systems that don't support symbolic links this option behaves exactly the same as the `stat` option.

`file mtime` *`fname`*
: Returns a decimal string giving the time at which file *`fname`* was last modified. The time is measured in the standard POSIX fashion as seconds from a fixed starting time (often January 1, 1970). If the file doesn't exist or its modified time cannot be queried then an error is generated.

`file owned` *`fname`*
: Returns `1` if file *`fname`* is owned by the current user, `0` otherwise.

`file readable` *`fname`*
: Returns `1` if file *`fname`* is readable by the current user, `0` otherwise.

`file readlink` *`fname`*
: Returns the value of the symbolic link given by *`fname`* (i.e. the name of the file it points to). If *`fname`* isn't a symbolic link or its value cannot be read, then an error is returned. On systems that don't support symbolic links this option is undefined.

`file rootname` *`fname`*
: Returns all of the characters in *`fname`* up to but not including the last '`.`' character in the *`fname`*. If *`fname`* doesn't contain a dot, then returns *`fname`*.

`file size` *`fname`*
: Returns a decimal string giving the size of file *`fname`* in bytes. If the file doesn't exist or its size cannot be queried then an error is generated.

`file stat` *fname varname*
: Invokes the stat kernel call on *fname*, and uses the variable given by *varname* to hold information returned from the kernel call. *varname* is treated as an array variable, and the following elements of that variable are set: `atime`, `ctime`, `dev`, `gid`, `ino`, `mode`, `mtime`, `nlink`, `size`, `type`, `uid`. Each element except `type` is a decimal string with the value of the corresponding field from the `stat` return structure; see the operating system reference manual entry for `stat` for details on the meanings of the values. The `type` element gives the type of the file in the same form returned by the command '`file type`'. This command returns an empty string.

`file tail` *fname*
: Returns all of the characters in *fname* after the last slash. If *fname* contains no slashes then returns *fname*.

`file type` *fname*
: Returns a string giving the type of file *fname*, which will be one of '`file`', '`directory`', '`characterSpecial`', '`blockSpecial`', '`fifo`', '`link`', or '`socket`'.

`file writable` *fname*
: Returns `1` if file *fname* is writable by the current user, `0` otherwise.

B.3.17 Flush buffered output for a file: `flush`

`flush` *fileid*

Flushes any output that has been buffered for *fileid*. *fileid* must be the return value from a previous call to `open`, or it may be `stdout` or `stderr` to access one of the standard I/O streams; it must refer to a file that was opened for writing. The command returns an empty string.

B.3.18 "For" loop: `for`

`for` *start tst nxt body*

`for` is a looping command, similar in structure to the C `for` statement. The *start*, *nxt*, and *body* arguments must be Tcl command strings, and *tst* is an expression string. The `for` command first invokes the Tcl interpreter to execute *start*. Then it repeatedly evaluates *tst* as an expression; if the result is non-zero it invokes the Tcl interpreter on *body*, then invokes the Tcl interpreter on *nxt*, then repeats the loop. The command terminates when *tst* evaluates to `0`. If a `continue` command is invoked within *body* then any remaining commands in the current execution of *body* are skipped; processing continues by invoking the Tcl interpreter on `nxt`, then evaluating `tst`, and so on. If a `break` command is invoked within *body* or `nxt`, then the `for` command will return immediately. The operation of `break` and `continue` are similar to the corresponding statements in C. `for` returns an empty string.

B.3.19 Iterate over all elements in a list: `foreach`

`foreach` *varname list body*

In this command *varname* is the name of a variable, *list* is a list of values to assign to *varname*, and *body* is a Tcl script. For each element of *list* (in order from left to right), `foreach` assigns the contents of the field to *varname* as if the `lindex` command had been used to extract the field, then calls the Tcl interpreter to execute *body*. The `break` and `continue` statements may be invoked inside *body*, with the same effect as in the `for` command. `foreach` returns an empty string.

B.3.20 Format in the style of `sprintf`: `format`

`format` *fstr* `[` *arg* `... ]`

This command generates a formatted string in the same way as the ANSI C `sprintf` procedure (it uses `sprintf` in its implementation). *fstr* indicates how to format the result, using '`%`' conversion specifiers as in `sprintf`, and the additional arguments, if any, provide values to be substituted into the result. The return value from `format` is the formatted string.

Details on formatting. The command operates by scanning *fstr* from left to right. Each character from the format string is appended to the result string unless it is a percent sign. If the character is a '`%`' then it is not copied to the result string. Instead, the characters following the '`%`' character are treated as a conversion specifier. The conversion specifier controls the conversion of the next successive *arg* to a particular format and the result is appended to the result string in place of the conversion specifier. If there are multiple conversion specifiers in the format string, then each one controls the conversion of one additional *arg*. The `format` command must be given enough *arg*s to meet the needs of all of the conversion specifiers in *fstr*.

Each conversion specifier may contain up to six different parts: an XPG3[1] position specifier, a set of flags, a minimum field width, a precision, a length modifier, and a conversion character. Any of these fields may be omitted except for the conversion character. The fields that are present

[1] XPG3 refers to the third edition of the *X/ Open Portability Guide*, an X/Open interface standard.

must appear in the order given above. The paragraphs below discuss each of these fields in turn.

If the '%' is followed by a decimal number and a '$', as in '%2$d', then the value to convert is not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where 1 corresponds to the first *arg*. If the conversion specifier requires multiple arguments because of * characters in the specifier then successive arguments are used, starting with the argument given by the number. This follows the XPG3 conventions for positional specifiers. If there are *any* positional specifiers in *fstr*, then *all* of the specifiers must be positional.

The second portion of a conversion specifier may contain any of the following flag characters, in any order:

- Specifies that the converted argument should be left-justified in its field (numbers are normally right-justified with leading spaces if needed).

+ Specifies that a number should always be printed with a sign, even if positive.

SPACE Specifies that a space should be added to the beginning of the number if the first character isn't a sign.

0 Specifies that the number should be padded on the left with zeroes instead of spaces.

Requests an alternate output form. For 'o' and 'O' conversions it guarantees that the first digit is always '0'. For 'x' or 'X' conversions, '0x' or '0X' (respectively) will be added to the beginning of the result unless it is zero. For all floating-point conversions ('e', 'E', 'f', 'g', and 'G') it guarantees that the result always has a decimal point. For 'g' and 'G' conversions it specifies that trailing zeroes should not be removed.

The third portion of a conversion specifier is a number giving a minimum field width for this conversion. It is typically used to make columns line up in tabular printouts. If the converted argument contains fewer characters than the minimum field width then it will be padded so that it is as wide as the minimum field width. Padding normally occurs by adding extra spaces on the left of the converted argument, but the '0' and '-' flags may be used to specify padding with zeroes on the left or with spaces on the right, respectively. If the minimum field width is specified as '*' rather than a number, then the next argument to the format command determines the minimum field width; it must be a numeric string.

The fourth portion of a conversion specifier is a precision, which consists of a period followed by a number. The number is used in different ways for different conversions. For 'e', 'E', and 'f' conversions it specifies the number of digits to appear to the right of the decimal point. For 'g' and 'G' conversions it specifies the total number of digits to appear, including those on both sides of the decimal point (however, trailing zeroes after the decimal point will still be omitted unless the # flag has been specified). For integer conversions, it specifies a mimimum number of digits to print (leading zeroes will be added if necessary). For s conversions it specifies the maximum number of characters to be printed; if the string is longer than this then the trailing characters will be dropped. If the precision is specified with '*' rather than a number then the next argument to the `format` command determines the precision; it must be a numeric string.

The fourth part of a conversion specifier is a length modifier, which must be 'h' or 'l'. If it is 'h' it specifies that the numeric value should be truncated to a 16-bit value before converting. This option is rarely useful. The 'l' modifier is ignored.

The last thing in a conversion specifier is an alphabetic character that determines what kind of conversion to perform. The following conversion characters are currently supported:

`d` Convert integer to signed decimal string.

`u` Convert integer to unsigned decimal string.

`i` Convert integer to signed decimal string; the integer may either be in decimal, in octal (with a leading '0') or in hexadecimal (with a leading '0x').

`o` Convert integer to unsigned octal string.

`x`
`X` Convert integer to unsigned hexadecimal string, using digits '0123456789abcdef' for 'x' and '0123456789ABCDEF' for 'X').

`c` Convert integer to the 8-bit character it represents.

`s` No conversion; just insert string.

`f` Convert floating-point number to signed decimal string of the form '*xx*.*yyy*', where the number of *y* digits is determined by the precision (default: 6). If the precision is 0 then no decimal point is output.

`e`
`E` Convert floating-point number to scientific notation in the form '*x*.*yyy*e+*zz*', where the number of *y* digits is determined by the precision (default: 6). If the precision is 0 then no decimal point is output. If the 'E' form is used then 'E' is printed instead of 'e'.

`g`
`G` If the exponent is less than -4 or greater than or equal to the precision, then convert floating-point number as for '`%e`' or '`%E`'. Otherwise convert as for '`%f`'. Trailing zeroes and a trailing decimal point are omitted.

`%` No conversion: just insert '`%`'.

For the numerical conversions the argument being converted must be an integer or floating-point string; `format` converts the argument to binary and then converts it back to a string according to the conversion specifier.

Differences from ANSI `sprintf`. The behavior of the `format` command is the same as the ANSI C `sprintf` procedure except for these differences:

1. '`%p`' and '`%n`' specifiers are not currently supported.
2. For '`%c`' conversions the argument must be a decimal string, which will then be converted to the corresponding character value.
3. The '`l`' modifier is ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the '`l`' modifier were present (i.e. type `double` is used for the internal representation). If the '`h`' modifier is specified then integer values are truncated to `short` before conversion.

B.3.21 Read a line from a file: `gets`

`gets` *`fileid`* `[` *`varname`* `]`

This command reads the next line from the file given by *`fileid`* and discards the terminating newline character. If *`varname`* is specified then the line is placed in the variable by that name and the return value is a count of the number of characters read (not including the newline). If the end of the file is reached before reading any characters then '`-1`' is returned and *`varname`* is set to an empty string. If *`varname`* is not specified then the return value will be the line (minus the newline character) or an empty string if the end of the file is reached before reading any characters. An empty string will also be returned if a line contains no characters except the newline, so `eof` may have to be used to determine what really happened. If the last character in the file is not a newline character then `gets` behaves as if there were an additional newline character at the end of the file. *`fileid`* must be `stdin` or the return value from a previous call to `open`; it must refer to a file that was opened for reading. Any existing end-of-file or error condition on the file is cleared at the beginning of the `gets` command.

B.3.22 Filename wildcard expansion: `glob`

`glob [ opts ] ptrn [ ptrn ... ]`

This command performs file name wildcard expansion ("globbing") in a fashion similar to the `csh` shell. It returns a list of the files whose names match any of the patterns specified as *ptrn* arguments.

If the initial arguments to glob start with '–' then they are treated as options. The following options are currently supported:

`-nocomplain`
: Allows an empty list to be returned without error; without this option an error is returned if the result list would be empty.

`--`
: Marks the end of *opts*. The argument following this one will be treated as a *ptrn* even if it starts with a '–'.

The *ptrn* arguments may contain any of the following special characters:

`?`
: Matches any single character.

`*`
: Matches any sequence of zero or more characters.

`[chars]`
: Matches any single character in *chars*. If *chars* contains a sequence of the form '*a*–*b*' then any character between *a* and *b* (inclusive), in your system's standard collating sequence—usually ASCII—will match.

`\x`
: Matches the character *x*.

`{a,b,...}`
: Matches any of the strings *a*, *b*, etc.

As with `csh`, a '.' at the beginning of a filename or just after a '/' must be matched explicitly or with a `{...}` construct. In addition, all '/' characters must be matched explicitly.

If the first character in a pattern is '~' then it refers to the home directory for the user whose name follows the '~'. If the '~' is followed immediately by '/' then the value of the HOME environment variable is used.

The `glob` command differs from `csh` wildcard expansion in two ways. First, it does not sort its result list (use the `lsort` command if you want the list sorted). Second, `glob` only returns the names of files that actually exist; in `csh` no check for existence is made unless a pattern contains a '?', '*', or '`[...]`' construct.

B.3.23 Access global variables: `global`

`global` *`varname`* `[` *`varname`* `... ]`

This command is ignored unless a Tcl procedure is being interpreted. If so then it declares each given *`varname`* listed to be a global variable, rather than a local one. For the duration of the current procedure (and only while executing in the current procedure), any reference to any *`varname`* will refer to the global variable by the same name.

B.3.24 Manipulate the history list: `history`

`history [` *`operation`* `] [` *`arg`* `... ]`

The history command performs one of several operations related to recently-executed commands recorded in a history list. Each of these recorded commands is referred to as an *event*. When specifying an event to the history command, the following forms may be used:

1. A number: if positive, it refers to the event with that number (all events are numbered starting at 1). If the number is negative, it selects an event relative to the current event (`-1` refers to the previous event, `-2` to the one before that, and so on).
2. A string: selects the most recent event that matches the string. An event is considered to match the string either if the string is the same as the first characters of the event, or if the string matches the event in the sense of the string match command.

B.3.24.1 History operations

The history command can take any of the following forms:

`history`
Same as history info, described below.

`history` add command [exec]
Adds the command argument to the history list as a new event. If `exec` is specified (or abbreviated) then the command is also executed and its result is returned. If `exec` isn't specified then an empty string is returned as result.

`history` change *`newval`* [event]
Replaces the value recorded for an event with *`newval`*. Event specifies the event to replace, and defaults to the current event (not event `-1`). This command is intended for use in commands that implement new forms of history substitution and wish to replace the current event (which invokes the substitution) with the command created through substitution. The return value is an empty string.

`history event` [`event`]
: Returns the value of the event given by event. Event defaults to `-1`. This command causes history revision to occur. See Section B.3.24.2 "History revision," page 113, for details.

`history info` [*n*]
: Returns a formatted string (intended for humans to read) giving the event number and contents for each of the events in the history list except the current event. If *n* is specified then only the most recent *n* events are returned.

`history keep` *n*
: This command may be used to change the size of the history list to *n* events. Initially, 20 events are retained in the history list. This command returns an empty string.

`history nextid`
: Returns the number of the next event to be recorded in the history list. It is useful for things like printing the event number in command-line prompts.

`history redo` [*evt*]
: Re-executes the command indicated by *evt* and return its result. *evt* defaults to `-1`. This command results in history revision. See Section B.3.24.2 "History revision," page 113, for details.

`history substitute` *old new* [*evt*]
: Retrieves the command given by *evt* (`-1` by default), replace any occurrences of *old* by *new* in the command (only simple character equality is supported; no wild cards), execute the resulting command, and return the result of that execution. This command results in history revision. See Section B.3.24.2 "History revision," page 113, for details.

`history words` *sel* [*evt*]
: Retrieves from the command given by *evt* (`-1` by default) the words given by *sel*, and return those words in a string separated by spaces. The *sel* argument has three forms. If it is a single number then it selects the word given by that number (`0` for the command name, `1` for its first argument, and so on). If it consists of two numbers separated by a dash, then it selects all the arguments between those two. Otherwise *sel* is treated as a pattern; all words matching that pattern (in the sense of '`string match`') are returned. In the numeric forms '`$`' may be used to select the last word of a command. For example, suppose the most recent command in the history list is

```
format {%s is %d years old} Alice [expr $ageInMonths/12]
```

Here are some history commands and the results they would produce (the result of each command is shown indented below it):

```
history words $
   [expr $ageInMonths/12]

history words 1-2
   {%s is %d years  old} Alice

history words *a*o*
   {%s is %d years old} [expr $ageInMonths/12]
```

'history words' results in history revision.

B.3.24.2 History revision

The history options event, redo, substitute, and words result in *history revision*. When one of these options is invoked then the current event is modified to eliminate the history command and replace it with the result of the history command. For example, suppose that the most recent command in the history list is

```
set a [expr $b+2]
```

and suppose that the next command invoked is one of the ones on the left side of the table below. The command actually recorded in the history event will be the corresponding one on the right side of the table.

```
history redo          set a [expr $b+2]
history s a b         set b [expr $b+2]
set c [history w 2]   set c [expr $b+2]
```

History revision is needed because event specifiers like -1 are only valid at a particular time: once more events have been added to the history list a different event specifier would be needed. History revision occurs even when history is invoked indirectly from the current event (e.g. a user types a command that invokes a Tcl procedure that invokes history): the top-level command whose execution eventually resulted in a history command is replaced. If you wish to invoke commands like 'history words' without history revision, you can use 'history event' to save the current history event and then use 'history change' to restore it later.

B.3.25 Execute scripts conditionally: `if`

```
if expr1 [ then ] body1
   elseif expr2 [ then ] body2
   elseif  . . .
   [ else ] [ bodyN ]
```

The `if` command evaluates *expr1* as an expression (in the same way that `expr` evaluates its argument). The value of the expression must be a Boolean (a numeric value, where 0 is false and anything is true, or a string value such as '`true`' or '`yes`' for true, and '`false`' or '`no`' for false); if it is true then *body1* is executed by passing it to the Tcl interpreter. Otherwise *expr2* is evaluated as an expression and if it is true then *body2* is executed, and so on. If none of the expressions evaluates to true then *bodyN* is executed. The `then` and `else` arguments are optional "noise words" to make the command easier to read. There may be any number of `elseif` clauses, including zero. *bodyN* may also be omitted as long as `else` is omitted too. The return value from the command is the result of the body script that was executed, or an empty string if none of the expressions was non-zero and there was no *bodyN*.

B.3.26 Increment the value of a variable: `incr`

`incr` *varname* `[` *n* `]`

Increments the value stored in the variable whose name is *varname*. The value of the variable must be an integer. If *n* is supplied then its value (which must also be an integer) is added to the value of variable *varname*; otherwise 1 is added to *varname*. The new value is stored as a decimal string in variable *varname* and also returned as result.

B.3.27 Report on state of Tcl interpreter: `info`

`info` *op* `[` *arg* `. . . ]`

This command provides information about various internals of the Tcl interpreter. Recognized values for *op* (which may be abbreviated) are:

`info args` *procname*
: Returns a list containing the names of the arguments to procedure *procname*, in order. *procname* must be the name of a Tcl command procedure.

`info body` *procname*
: Returns the body of procedure *procname*. *procname* must be the name of a Tcl command procedure.

`info cmdcount`
: Returns a count of the total number of commands that have been invoked in this interpreter.

`info commands` [*ptn*]
: If *ptn* isn't specified, returns a list of names of all the Tcl commands, including both the built-in commands written in C and the command procedures defined using the `proc` command. If *ptn* is specified, only those names matching *ptn* are returned. Matching is determined using the same rules as for '`string match`'.

`info complete` *cmd*
: Returns `1` if *cmd* is a complete Tcl command in the sense of having no unclosed quotes, braces, brackets or array element names. If the command doesn't appear to be complete then `0` is returned. This command is typically used in line-oriented input environments to allow users to type in commands that span multiple lines; if the command isn't complete, the script can delay evaluating it until additional lines have been typed to complete the command.

`info default` *procname arg varname*
: *procname* must be the name of a Tcl command procedure and *arg* must be the name of an argument to that procedure. If *arg* doesn't have a default value then the command returns `0`. Otherwise it returns `1` and places the default value of *arg* into variable *varname*.

`info exists` *varname*
: Returns `1` if the variable named *varname* exists in the current context (either as a global or local variable); returns `0` otherwise.

`info globals` [*ptn*]
: If *ptn* isn't specified, returns a list of all the names of currently-defined global variables. If pattern is specified, only those names matching *ptn* are returned. Matching is determined using the same rules as for '`string match`'.

`info level` [*n*]
: If *n* is not specified, this command returns a number giving the stack level of the invoking procedure, or `0` if the command is invoked at top-level. If *n* (a number) is specified, then the result is a list consisting of the name and arguments for the procedure call at level *n* on the stack. If *n* is positive then it selects a particular stack level (`1` refers to the top-most active procedure, `2` to the procedure it called, and so

on); otherwise it gives a level relative to the current level (`0` refers to the current procedure, `-1` to its caller, and so on). See Section B.3.60 “Different stack frame: `uplevel`,” page 139, for more information on what stack levels mean.

`info library`
: Returns the name of the library directory in which standard Tcl scripts are stored. The default value for the library is compiled into Tcl, but it may be overridden by setting the `TCL_LIBRARY` environment variable. If there is no `TCL_LIBRARY` variable and no compiled-in value then an error is generated. See Section B.5 “Tcl standard library,” page 143, for details of the facilities provided by the Tcl script library. Normally each application will have its own application-specific script library in addition to the Tcl script library; I suggest that each application set a global variable with a name like `app_library` (where *app* is the application's name) to hold the location of that application's library directory.

`info locals [` *ptn* `]`
: If *ptn* isn't specified, returns a list of all the names of currently-defined local variables, including arguments to the current procedure, if any. Variables defined with the `global` and `upvar` commands will not be returned. If *ptn* is specified, only those names matching *ptn* are returned. Matching is determined using the same rules as for ‘`string match`’.

`info patchlevel`
: Returns a decimal integer giving the current patch level for Tcl. The patch level is incremented for each new release or patch, and it uniquely identifies an official version of Tcl.

`info procs [` *ptn* `]`
: If *ptn* isn't specified, returns a list of all the names of Tcl command procedures (only those defined with `proc`; not built-in commands). If *ptn* is specified, only those names matching *ptn* are returned. Matching is determined using the same rules as for ‘`string match`’.

`info script`
: If a Tcl script file is currently being evaluated, then this command returns the name of the innermost file being processed. Otherwise the command returns an empty string.

`info tclversion`
: Returns the version number for this version of Tcl in the form ‘*x*.*y*’, where changes to *x* represent major changes with probable incompatibilities and changes to *y* represent small

enhancements and bug fixes that retain backward compatibility.

`info vars` [*ptn*]
: If *ptn* isn't specified, returns a list of all the names of currently-visible variables, including both locals and currently-visible globals. If *ptn* is specified, only those names matching *ptn* are returned. Matching is determined using the same rules as for '`string match`'.

B.3.28 Join together list elements: `join`

`join` *lst* [*delim*]

The *lst* argument must be a valid Tcl list. This command returns the string formed by joining all of the elements of *lst* together with the string *delim* separating each adjacent pair of elements. The *delim* argument defaults to a space character.

B.3.29 Append list elements: `lappend`

`lappend` *varname val* [*val* ...]

This command treats the variable given by *varname* as a list and appends each of the *val* arguments to that list as a separate element, with spaces between elements. If *varname* doesn't exist, it is created as a list with elements given by the *val* arguments. `lappend` is similar to `append` except that the values are appended as list elements rather than raw text. This command provides a relatively efficient way to build up large lists. For example, '`lappend a $b`' is much more efficient than '`set a [concat $a [list $b]]`' when '`$a`' is long.

B.3.30 Retrieve an element from a list: `lindex`

`lindex` *lst ix*

This command treats *lst* as a Tcl list and returns the element indexed by *ix* from it (`0` refers to the first element of the list). In extracting the element, `lindex` observes the same rules concerning braces and quotes and backslashes as the Tcl command interpreter; however, variable substitution and command substitution do not occur. If *ix* is negative or greater than or equal to the number of elements in *lst*, then an empty string is returned.

B.3.31 Insert elements into a list: `linsert`

`linsert` *`lst ix elt`* [*`elt`* ...]

This command produces a new list from *`lst`* by inserting all of the *`elt`* arguments just before the element at position *`ix`* of *`lst`*. Each *`elt`* argument will become a separate element of the new list. If *`ix`* is less than or equal to zero, then the new elements are inserted at the beginning of the list. If *`ix`* is greater than or equal to the number of elements in *`lst`*, then the new elements are appended to the list.

B.3.32 Create a list: `list`

`list` [*`arg`* ...]

This command returns a list composed of all the *`arg`* values, or an empty string if no *`arg`* is specified. Braces and backslashes get added as necessary, so that the `index` command may be used on the result to re-extract the original arguments, and also so that `eval` may be used to execute the resulting list, with the first *`arg`* comprising the command's name and the other args comprising its arguments. `list` produces slightly different results than `concat`: `concat` removes one level of grouping before forming the list, while `list` works directly from the original arguments. For example, the command

```
list a b {c d e} {f {g h}}
```

will return

```
a b {c d e} {f {g h}}
```

while `concat` with the same arguments will return

```
a b c d e f {g h}
```

B.3.33 Number of elements in a list: `llength`

`llength` *`lst`*

Treats *`lst`* as a list and returns a decimal string giving the number of elements in it.

B.3.34 Adjacent elements from a list: `lrange`

`lrange` *`lst first last`*

`lst` must be a valid Tcl list. This command will return a new list consisting of elements *`first`* through *`last`*, inclusive. *`last`* may be '`end`' (or any abbreviation of it) to refer to the last element of the list. If *`first`* is less than zero, it is treated as if it were zero. If *`last`* is greater than or equal to the number of elements in the list, then it is treated as if it were '`end`'. If *`first`* is greater than *`last`* then an empty string is returned. Note: '`lrange` *`lst first first`*' does not always produce the same result as '`lindex` *`lst first`*' (although it often does for simple fields that aren't enclosed in braces); it does, however, produce exactly the same results as '`list [lindex` *`lst first`*`]`'.

B.3.35 Replace elements in a list: `lreplace`

`lreplace` *`lst first last`* [*`elt`* ...]

`lreplace` returns a new list formed by replacing one or more elements of *`lst`* with the *`elt`* arguments. *`first`* gives the index in *`lst`* of the first element to be replaced. If *`first`* is less than zero then it refers to the first element of *`lst`*; the element indicated by *`first`* must exist in the list. *`last`* gives the index in *`lst`* of the last element to be replaced; it must be greater than or equal to *`first`*. *`last`* may be end (or any abbreviation of it) to indicate that all elements between *`first`* and the end of the list should be replaced. The *`elt`* arguments specify zero or more new arguments to be added to the list in place of those that were deleted. Each *`elt`* argument will become a separate element of the list. If no *`elt`* arguments are specified, then the elements between *`first`* and *`last`* are simply deleted.

B.3.36 Search for element in list: `lsearch`

`lsearch` [*`mode`*] *`lst ptn`*

This command searches the elements of the list *`lst`* to see if one of them matches *`ptn`*. If so, the command returns the index of the first matching element. If not, the command returns `-1`. The *`mode`* argument indicates how the elements of the list are to be matched against *`ptn`* and it must have one of the following values:

`-exact` The list element must contain exactly the same string as *`ptn`*.

`-glob` *`ptn`* is a glob-style pattern which is matched against each list element using the same rules as the '`string match`' command.

`-regexp` *ptn* is treated as a regular expression and matched against each list element using the same rules as the `regexp` command.

If *mode* is omitted then it defaults to '`-glob`'.

B.3.37 Sort the elements of a list: `lsort`

`lsort [ opts ] lst`

This command sorts the elements of a list *lst*, returning a new list in sorted order. By default ASCII sorting is used with the result returned in increasing order. However, any of the following options may be specified before *lst* to control the sorting process (unique abbreviations are accepted):

`-ascii` Use string comparison with ASCII collation order. This is the default.

`-integer` Convert list elements to integers and use integer comparison.

`-real` Convert list elements to floating-point values and use floating comparison.

`-command cmd`
: Use *cmd* as a comparison command. To compare two elements, evaluate a Tcl script consisting of *cmd* with the two elements appended as additional arguments. The script should return an integer less than, equal to, or greater than zero if the first element is to be considered less than, equal to, or greater than the second, respectively.

`-increasing`
: Sort the list in increasing order ("smallest" items first). This is the default.

`-decreasing`
: Sort the list in decreasing order ("largest" items first).

B.3.38 Open a file: `open`

`open` *fname* [*access*] [*perm*]

This command opens a file and returns an identifier that may be used in future invocations of commands like `read`, `puts`, and `close`. *fname* gives the name of the file to open; if it starts with a tilde then tilde substitution is performed (see Section B.3.6 "Change working directory: `cd`," page 98). If the first character of *fname* is '|' then the remaining characters of *fname* are treated as a command pipeline to invoke, in the same style as for `exec`. In this case, the identifier returned by `open` may be used to write to the command's input pipe or read from its output pipe.

The *access* argument indicates the way in which the file (or command pipeline) is to be accessed. It may take two forms, either a string in the form that would be passed to the `fopen` library procedure or a list of POSIX access flags. It defaults to '`r`'. In the first form *access* may have any of the following values:

`r` Open the file for reading only; the file must already exist.

`r+` Open the file for both reading and writing; the file must already exist.

`w` Open the file for writing only. Truncate it if it exists. If it doesn't exist, create a new file.

`w+` Open the file for reading and writing. Truncate it if it exists. If it doesn't exist, create a new file.

`a` Open the file for writing only. The file must already exist, and the file is positioned so that new data is appended to the file.

`a+` Open the file for reading and writing. If the file doesn't exist, create a new empty file. Set the initial access position to the end of the file.

In the second form, *access* consists of a list of any of the following flags, all of which have the standard POSIX meanings. One of the flags must be either `RDONLY`, `WRONLY` or `RDWR`.

`RDONLY` Open the file for reading only.

`WRONLY` Open the file for writing only.

`RDWR` Open the file for both reading and writing.

`APPEND` Set the file pointer to the end of the file prior to each write.

`CREAT` Create the file if it doesn't already exist (without this flag it is an error for the file not to exist).

`EXCL` If `CREAT` is specified also, an error is returned if the file already exists.

`NOCTTY` If the file is a terminal device, this flag prevents the file from becoming the controlling terminal of the process.

`NONBLOCK` Prevents the process from blocking while opening the file. For details refer to your system documentation on the `open` system call `O_NONBLOCK` flag.

`TRUNC` If the file exists it is truncated to zero length.

If a new file is created as part of opening it, *`perm`* (an integer) is used to set the permissions for the new file in conjunction with the process's file mode creation mask. *`perm`* defaults to `0666`.

If a file is opened for both reading and writing then `seek` must be invoked between a read and a write, or vice versa (this restriction does not apply to command pipelines opened with `open`). When *`fname`* specifies a command pipeline and a write-only access is used, then standard output from the pipeline is directed to the current standard output unless overridden by the command. When *`fname`* specifies a command pipeline and a read-only access is used, then standard input from the pipeline is taken from the current standard input unless overridden by the command.

B.3.39 Retrieve process ids: `pid`

`pid [` *`fileid`* `]`

If the *`fileid`* argument is given then it should normally refer to a process pipeline created with the open command. In this case the `pid` command will return a list whose elements are the process identifiers of all the processes in the pipeline, in order. The list will be empty if *`fileid`* refers to an open file that isn't a process pipeline. If no *`fileid`* argument is given then `pid` returns the process identifier of the current process. All process identifiers are returned as decimal strings.

B.3.40 Create a Tcl procedure: `proc`

`proc` *`pname arglist body`*

The `proc` command creates a new Tcl procedure named *`pname`*, replacing any existing command or procedure there may have been by that name. Whenever the new command is invoked, the contents of *`body`* will be executed by the Tcl interpreter. *`arglist`* specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list, with either one or two fields. If there is only a single field in the specifier then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value.

When *pname* is invoked a local variable will be created for each of the formal arguments to the procedure; its value will be the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments. There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name `args`, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case, all of the actual arguments starting at the one that would be assigned to `args` are combined into a list (as if the list command had been used); this combined value is assigned to the local variable `args`.

When *body* is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can only be accessed by invoking the `global` command or the `upvar` command.

The `proc` command returns an empty string. When a procedure is invoked, the procedure's result is the value specified in a `return` command. If the procedure doesn't execute an explicit `return`, then its result is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure as a whole will return that same error.

B.3.41 Write to a file: `puts`

`puts [ -nonewline ] [ `*`fileid`*` ] `*`str`*

Writes the characters given by *str* to the file given by *fileid*. *fileid* must have been the return value from a previous call to `open`, or it may be `stdout` or `stderr` to refer to one of the standard I/O channels; it must refer to a file that was opened for writing. If no *fileid* is specified then it defaults to `stdout`. `puts` normally outputs a newline character after *str*, but this feature may be suppressed by specifying the '`-nonewline`' switch. Output to files is buffered internally by Tcl; the `flush` command may be used to force buffered characters to be output.

B.3.42 Current working directory: `pwd`

`pwd`

Returns the path name of the current working directory.

B.3.43 Read from a file: `read`

`read [ -nonewline ]` *`fileid`*
`read` *`fileid numbytes`*

In the first form, all of the remaining bytes are read from the file given by *`fileid`*; they are returned as the result of the command. If the '`-nonewline`' option is specified then the last character of the file is discarded if it is a newline. In the second form, the extra argument specifies how many bytes to read; exactly this many bytes will be read and returned, unless there are fewer than *`numbytes`* bytes left in the file; in this case, all the remaining bytes are returned. *`fileid`* must be `stdin` or the return value from a previous call to `open`; it must refer to a file that was opened for reading. Any existing end-of-file or error condition on the file is cleared at the beginning of the read command.

B.3.44 Match a regular expression: `regexp`

`regexp [` *`opts`* `]` *`rexp str`* `[` *`mvar`* `] [` *`subvar`* `... ]`

Determines whether the regular expression *`rexp`* matches part or all of the string *`str`* and returns `1` if it does, `0` if it doesn't.

If additional arguments are specified after *`str`* then they are treated as the names of variables in which to return information about which parts of *`str`* matched *`rexp`*. *`mvar`* will be set to the range of *`str`* that matched all of *`rexp`*. The first *`subvar`* will contain the characters in *`str`* that matched the leftmost parenthesized subexpression within *`rexp`*, the next *`subvar`* will contain the characters that matched the next parenthesized subexpression to the right in *`rexp`*, and so on.

If the initial arguments to `regexp` start with '`-`' then they are treated as options. The following options are currently supported:

`-nocase`
: Causes upper-case characters in *`str`* to be treated as lower case during the matching process.

`-indices`
: Changes what is stored in each *`subvar`*. Instead of storing the matching characters from *`str`*, each variable will contain a list of two decimal strings giving the indices in *`str`* of the first and last characters in the matching range of characters.

`--`
: Marks the end of *`opts`*. The argument following this one will be treated as *`rexp`* even if it starts with a '`-`'.

If there are more names in the *`subvar`* list than there are parenthesized subexpressions within *`rexp`*, or if a particular subexpression in *`rexp`* doesn't match the string (e.g. because it was in a portion of the expression that wasn't matched), then the corresponding *`subvar`* will be set to '`-1 -1`' if '`-indices`' has been specified or to an empty string otherwise.

B.3.44.1 Regular expressions

Regular expressions are implemented using Henry Spencer's package (thanks, Henry!), and much of the description of regular expressions below is copied verbatim from his reference manual entry.

A regular expression is zero or more branches, separated by '|'. It matches anything that matches one of the branches.

A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by '*', '+', or '?'. An atom followed by '*' matches a sequence of 0 or more matches of the atom. An atom followed by '+' matches a sequence of 1 or more matches of the atom. An atom followed by '?' matches a match of the atom, or the empty string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a range (see below), '.' (matching any single character), '^' (matching the empty string at the beginning of the input string), '$' (matching the empty string at the end of the input string), a '\' followed by a single character (matching that character), or a single character with no other significance (matching that character).

A range is a sequence of characters enclosed in '[]'. It normally matches any single character from the sequence. If the sequence begins with '^', it matches any single character not from the rest of the sequence. If two characters in the sequence are separated by '-', this is shorthand for the full list of ASCII characters between them (e.g. '[0-9]' matches any decimal digit). To include a literal ']' in the sequence, make it the first character (following a possible '^'). To include a literal '-', make it the first or last character.

B.3.44.2 Choosing among matches

In general there may be more than one way to match a regular expression to an input string. For example, consider the command

```
regexp (a*)b* aabaaabb x y
```

Considering only the rules given so far, `x` and `y` could end up with the values 'aabb' and 'aa', 'aaab' and 'aaa', 'ab' and 'a', or any of several other combinations. To resolve this potential ambiguity regexp chooses among alternatives using the rule "first then longest". In other words, it consders the possible matches in order working from left to right across the input string and the pattern, and it attempts to match longer pieces of the input string before shorter ones. More specifically, the following rules apply in decreasing order of priority:

1. If a regular expression could match two different parts of an input string then it will match the one that begins earliest.

2. If a regular expression contains '|' characters then the leftmost matching sub-expression is chosen.
3. In '*', '+', and '?' constructs, longer matches are chosen in preference to shorter ones.
4. In sequences of expression components the components are considered from left to right.

In the example above, '(a*)b*' matches 'aab': the '(a*)' portion of the pattern is matched first and it consumes the leading 'aa'; then the 'b*' portion of the pattern consumes the next 'b'. Or consider the following example:

```
regexp  (ab|a)(b*)c  abc  x  y  z
```

After this command x will be 'abc', y will be 'ab', and z will be an empty string. Rule 4 specifies that '(ab|a)' gets first shot at the input string and Rule 2 specifies that the 'ab' subexpression is checked before the 'a' sub-expression. Thus the 'b' has already been claimed before the '(b*)' component is checked and '(b*)' must match an empty string.

B.3.45 Regular expression substitutions: `regsub`

`regsub [ `*`opts`*` ] `*`rexp str repl varname`*

This command matches the regular expression *rexp* against *str*, and it copies *str* to the variable whose name is given by *varname*. The command returns 1 if there is a match and 0 if there isn't. If there is a match, then while copying *str* to *varname* the portion of *str* that matched *rexp* is replaced with *repl*. If *repl* contains a '&' or '\0', that character or string is replaced in the substitution with the portion of *str* that matched *rexp*. If *repl* contains a '\n', where *n* is a digit between 1 and 9, then it is replaced in the substitution with the portion of *str* that matched the *n*th parenthesized subexpression of *rexp*. Additional backslashes may be used in *repl* to prevent special interpretation of '&' or '\0' or '\n' or backslash. The use of backslashes in *repl* tends to interact badly with the Tcl parser's use of backslashes, so it's generally safest to enclose *repl* in braces if it includes backslashes.

If the initial arguments to `regsub` start with '-' then they are treated as options. The following options are currently supported:

`-all`
: All ranges in *str* that match *rexp* are found and substitution is performed for each of these ranges. Without this option only the first matching range is found and substituted. If '-all' is specified, then '&' and '\n' sequences are handled for each substitution using the information from the corresponding match.

`-nocase` Upper-case characters in *str* will be converted to lower-case before matching against *rexp*; however, substitutions specified by *repl* use the original unconverted form of *str*.

`--` Marks the end of *opts*. The argument following this one will be treated as *rexp* even if a '-' is its first character.

See Section B.3.44 "Match a regular expression: regexp," page 124, for details on the interpretation of regular expressions.

B.3.46 Rename or delete a command: `rename`

`rename` *oldname newname*

Rename the command that used to be called *oldname* so that it is now called *newname*. If *newname* is an empty string then *oldname* is deleted. The `rename` command returns an empty string as result.

B.3.47 Return from a procedure: `return`

`return [ -code` *c* `] [ -errorinfo` *i* `] [ -errorcode` *e* `] [` *str* `]`

Return immediately from the current procedure (or top-level command or source command), with *str* as the return value. If *str* is not specified then an empty string will be returned as result.

Exceptional returns. In the usual case where the '`-code`' option isn't specified the procedure will return normally (its completion code will be `TCL_OK`). However, the '`-code` *c*' option may be used to generate an exceptional return from the procedure. *c* may have any of the following values:

`ok` Normal return: same as if the option is omitted.

`error` Error return: same as if the `error` command were used to terminate the procedure, except for handling of `errorInfo` and `errorCode` variables (see below).

`return` The current procedure will return with a completion code of `TCL_RETURN`, so that the procedure that invoked it will return also.

`break` The current procedure will return with a completion code of `TCL_BREAK`, which will terminate the innermost nested loop in the code that invoked the current procedure.

`continue` The current procedure will return with a completion code of `TCL_CONTINUE`, which will terminate the current iteration of the innermost nested loop in the code that invoked the current procedure.

`val` `val` must be an integer; it will be returned as the completion code for the current procedure.

The '`-code`' option is rarely used. It is provided so that procedures that implement new control structures can reflect exceptional conditions back to their callers.

Two additional options, '`-errorinfo`' and '`-errorcode`', may be used to provide additional information during error returns. These options are ignored unless you specify '`-code error`'.

The '`-errorinfo` *i*' option specifies an initial stack trace for the `errorInfo` variable; if it is not specified then the stack trace left in `errorInfo` will include the call to the procedure and higher levels on the stack but it will not include any information about the context of the error within the procedure. Typically the *i* value is supplied from the value left in `errorInfo` after a `catch` command trapped an error within the procedure.

If the '`-errorcode` *e*' option is specified then *e* provides a value for the `errorCode` variable. If the option is not specified then `errorCode` will default to '`NONE`'.

B.3.48 Parse string like `sscanf`: `scan`

`scan` *str fmt varname* [*varname* ...]

This command parses fields from an input string in the same fashion as the ANSI C `sscanf` procedure and returns a count of the number of fields sucessfully parsed. *str* gives the input to be parsed and *fmt* indicates how to parse it, using '`%`' conversion specifiers as in `sscanf`. Each *varname* gives the name of a variable; when a field is scanned from *str* the result is converted back into a string and assigned to the corresponding variable.

Details on scanning. `scan` operates by scanning *str* and *fmt* together. If the next character in *fmt* is a blank or tab then it is ignored. Otherwise, if it isn't a '`%`' character then it must match the next non-white-space character of *str*. When a '`%`' is encountered in *fmt*, it indicates the start of a conversion specifier. A conversion specifier contains up to three fields after the '`%`': a '`*`', which indicates that the converted value is to be discarded instead of assigned to a variable; a number indicating a maximum field width; and a conversion character. All of these fields are optional except for the conversion character.

When scan finds a conversion specifier in *fmt*, it first skips any white-space characters in *str*. Then it converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to `scan`. The following conversion characters are supported:

`d` The input field must be a decimal integer. It is read in and the value is stored in the variable as a decimal string.

`o` The input field must be an octal integer. It is read in and the value is stored in the variable as a decimal string.

`x` The input field must be a hexadecimal integer. It is read in and the value is stored in the variable as a decimal string.

`c` A single character is read in and its binary value is stored in the variable as a decimal string. Initial white space is not skipped in this case, so the input field may be a white-space character. This conversion is different from the ANSI standard in that the input field always consists of a single character and no field width may be specified.

`s` The input field consists of all the characters up to the next white-space character; the characters are copied to the variable.

`e`
`f`
`g` The input field must be a floating-point number consisting of an optional sign, a string of decimal digits possibly containing a decimal point, and an optional exponent consisting of an '`e`' or '`E`' followed by an optional sign and a string of decimal digits. It is read in and stored in the variable as a floating-point string.

`[`*`chars`*`]` The input field consists of any number of the characters in *chars*. The matching string is stored in the variable. If the first character between the brackets is a '`]`' then it is treated as part of chars rather than the closing bracket for the set.

`[^`*`chars`*`]` The input field consists of any number of characters not in *chars*. The matching string is stored in the variable. If the character immediately following the '`^`' is a '`]`' then it is treated as part of the set rather than the closing bracket for the set.

The number of characters read from the input for a conversion is the largest number that makes sense for that particular conversion (e.g. as many decimal digits as possible for '`%d`', as many octal digits as possible for '`%o`', and so on). The input field for a given conversion terminates either when a white-space character is encountered or when the maximum field width has been reached, whichever comes first. If a '`*`' is present in the conversion specifier then no variable is assigned and the next scan argument is not consumed.

Differences from ANSI `sscanf`. The behavior of the `scan` command is the same as the behavior of the ANSI C `sscanf` function except for these differences:

1. '`%p`' and '`%n`' conversion specifiers are not currently supported.
2. For '`%c`' conversions a single character value is converted to a decimal string, which is then assigned to the corresponding *varname*; no field width may be specified for this conversion.
3. The '`l`', '`h`', and '`L`' modifiers are ignored; integer values are always converted as if there were no modifier present and real values are always converted as if the '`l`' modifier were present (i.e. type `double` is used for the internal representation).

B.3.49 Access position for an open file: `seek`

`seek` *fileid offset* [*origin*]

Change the current access position for *fileid*. *fileid* must be the return value from a previous call to `open`, or it may be `stdin`, `stdout`, or `stderr` to refer to one of the standard I/O channels. The *offset* and *origin* arguments specify the position at which the next read or write will occur for *fileid*. *offset* must be an integer (which may be negative) and *origin* must be one of the following:

`start`
: The new access position will be *offset* bytes from the start of the file.

`current`
: The new access position will be *offset* bytes from the current access position; a negative *offset* moves the access position backwards in the file.

`end`
: The new access position will be *offset* bytes from the end of the file. A negative *offset* places the access position before the end-of-file, and a positive *offset* places the access position after the end-of-file.

The *origin* argument defaults to `start`. This command returns an empty string.

B.3.50 Read and write variables: `set`

`set` *`varname`* `[` *`val`* `]`

Returns the value of variable *`varname`*. If *`val`* is specified, then set the value of *`varname`* to *`val`*, creating a new variable if one doesn't already exist, and return its value. If *`varname`* contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the first open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise *`varname`* refers to a scalar variable. If no procedure is active, then *`varname`* refers to a global variable. If a procedure is active, then *`varname`* refers to a parameter or local variable of the procedure unless the global command has been invoked to declare *`varname`* to be global.

B.3.51 Evaluate a file as a Tcl script: `source`

`source` *`fname`*

Read file *`fname`* and pass the contents to the Tcl interpreter as a script to evaluate in the normal fashion. The return value from source is the return value of the last command executed from the file. If an error occurs in evaluating the contents of the file then the source command will return that error. If a return command is invoked from within the file then the remainder of the file will be skipped and the source command will return normally with the result from the return command. If *`fname`* starts with a tilde, then it is tilde-substituted as described in Section B.3.6 "Change working directory: `cd`," page 98.

B.3.52 Split a string into a Tcl list: `split`

`split` *`str`* `[` *`splitchars`* `]`

Returns a list created by splitting a string *`str`* at each character that is in the *`splitchars`* argument. Each element of the result list will consist of the characters from *`str`* that lie between instances of the characters in *`splitchars`*. Empty list elements will be generated if *`str`* contains adjacent characters in *`splitchars`*, or if the first or last character of *`str`* is in *`splitchars`*. If *`splitchars`* is an empty string then each character of *`str`* becomes a separate element of the result list. *`splitchars`* defaults to the standard white-space characters. For example,

```
split "comp.unix.misc" .
```

returns '`comp unix misc`' and

```
split "Hello world" {}
```

returns '`H e l l o { } w o r l d`'.

B.3.53 Manipulate strings: `string`

`string` *op arg* [*arg* ...]

Performs one of several string operations, depending on *op*. The recognized alternatives for *op* (which may be abbreviated) are:

`string compare` *str1 str2*
: Perform a character-by-character comparison of strings *str1* and *str2* in the same way as the C `strcmp` procedure. Return `-1`, `0`, or `1`, depending on whether *str1* is lexicographically less than, equal to, or greater than *str2*.

`string first` *str1 str2*
: Search the string *str2* for a sequence of characters that exactly match the characters in the string *str1*. If found, return the index of the first character in the first such match within *str2*. If not found, return `-1`.

`string index` *str charix*
: Returns the character number *charix* from the string *str*. A *charix* of 0 corresponds to the first character of the string. If *charix* is less than 0 or greater than or equal to the length of the string then an empty string is returned.

`string last` *str1 str2*
: Search the string *str2* for a sequence of characters that exactly match the characters in the string *str1*. If found, return the index of the first character in the last such match within *str2*. If there is no match, then return `-1`.

`string length` *str*
: Returns a decimal string giving the number of characters in *str*.

`string match` *ptn str*
: See if the pattern *ptn* matches the string *str*; return `1` if it does, `0` if it doesn't. Matching is done in a fashion similar to that used by the C-shell. For the two strings to match, their contents must be identical except that the following special sequences may appear in *ptn*:

  `*` — Matches any sequence of characters in *str*, including a null string.

  `?` — Matches any single character in *str*.

`[chars]` Matches any character in the string *chars*. If a sequence of the form *x*−*y* appears in *chars*, then any character between *x* and *y*, inclusive, will match.

`\x` Matches the single character *x*. This provides a way of avoiding the special interpretation of the characters '`*?[]\`' in *ptn*.

`string range` *str first last*
Returns a range of consecutive characters from *str*, starting with the character whose index is *first* and ending with the character whose index is *last*. An index of `0` refers to the first character of the string. *last* may be `end` (or any abbreviation of it) to refer to the last character of the string. If *first* is less than zero then it is treated as if it were zero, and if *last* is greater than or equal to the length of the string then it is treated as if it were `end`. If *first* is greater than *last* then an empty string is returned.

`string tolower` *str*
Returns a value equal to *str* except that all upper case letters have been converted to lower case.

`string toupper` *str*
Returns a value equal to *str* except that all lower case letters have been converted to upper case.

`string trim` *str* [*chars*]
Returns a value equal to *str* except that any leading or trailing characters from the string *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

`string trimleft` *str* [*chars*]
Returns a value equal to *str* except that any leading characters from the string *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

`string trimright` *str* [*chars*]
Returns a value equal to *str* except that any trailing characters from the string *chars* are removed. If *chars* is not specified then white space is removed (spaces, tabs, newlines, and carriage returns).

B.3.54 Choose a case to execute: `switch`

`switch [` *opts* `]` *str ptn body* `[` *ptn body* `... ]`
`switch [` *opts* `]` *str* `{`*ptn body* `[` *ptn body* `... ] }`

The `switch` command matches its *str* argument against each of the patterns in the *ptn* arguments, in the order specified. As soon as it finds a pattern that matches *str* it evaluates the following *body* argument by passing it recursively to the Tcl interpreter and returns the result of that evaluation. If the last *ptn* argument is `default` then it matches anything. If no *ptn* argument matches *str* and no `default` is given, then the `switch` command returns an empty string.

If the initial arguments to `switch` start with '–' then they are treated as options. The following options are currently recognized:

`-exact` Use exact matching when comparing *str* to a pattern. This is the default.

`-glob` When matching *str* to the patterns, use glob-style matching (i.e. the same as implemented by the '`string match`' command).

`-regexp` When matching *str* to the patterns, use regular expression matching (i.e. the same as implemented by the `regexp` command).

`--` Marks the end of *opts*. The argument following this one will be treated as *str* even if a '–' is its first character.

Two syntaxes are provided for the *ptn* and *body* arguments. The first uses a separate argument for each of the patterns and commands; this form is convenient if substitutions are desired on some of the patterns or commands. The second form places all of the patterns and commands together into a single argument; the argument must have proper list structure, with the elements of the list being the patterns and commands. The second form makes it easy to construct multi-line `switch` commands, since the braces around the whole list make it unnecessary to include a backslash at the end of each line. Since the *ptn* arguments are in braces in the second form, no command or variable substitutions are performed on them; this makes the behavior of the second form different than the first form in some cases.

If a *body* is specified as '–' it means that the body for the next pattern should also be used as the body for this pattern (if the next pattern also has a *body* of '–' then the body after that is used, and so on). This feature makes it possible to share a single body among several patterns.

Below are some examples of `switch` commands:

```
switch abc a - b {format 1} abc {format 2} default {format 3}
```

will return 2,

```
switch -regexp aaab {
  ^a.*b$ -
  b {format 1}
  a* {format 2}
  default {format 3}
}
```

will return 1, and

```
switch xyz {
  a
    -
  b
    {format 1}
  a*
    {format 2}
  default
    {format 3}
}
```

will return 3.

B.3.55 Access position for an open file: `tell`

`tell` *`fileid`*

Returns a decimal string giving the current access position in *`fileid`*. *`fileid`* must be the return value from a previous call to `open`, or it may be `stdin`, `stdout`, or `stderr` to refer to one of the standard I/O channels.

B.3.56 Measure elapsed time: `time`

`time` *`script`* `[` *`n`* `]`

This command will call the Tcl interpreter *n* times to evaluate *`script`* (or once if *n* isn't specified). It will then return a string of the form

```
503 microseconds per iteration
```

which indicates the average amount of time required per iteration, in microseconds. Time is measured in elapsed time, not CPU time.

B.3.57 Monitor variable use: `trace`

`trace` *`option`* [*`arg`* ...]

This command causes Tcl commands to be executed whenever certain operations are invoked. At present, only variable tracing is implemented. The recognized options (which may be abbreviated) are:

`trace variable` *`name ops cmd`*

Arrange for *`cmd`* to be executed whenever variable *`name`* is accessed in one of the ways given by *`ops`*. *`name`* may refer to a normal variable, an element of an array, or to an array as a whole (i.e. *`name`* may be just the name of an array, with no parenthesized index). If *`name`* refers to a whole array, then *`cmd`* is invoked whenever any element of the array is manipulated.

`ops` indicates which operations are of interest, and consists of one or more of the following letters:

`r` Invoke *`cmd`* whenever the variable is read.

`w` Invoke *`cmd`* whenever the variable is written.

`u` Invoke *`cmd`* whenever the variable is unset. Variables can be unset explicitly with the `unset` command, or implicitly when procedures return (all of their local variables are unset). Variables are also unset when interpreters are deleted, but traces will not be invoked then because there is no interpreter in which to execute them.

When the trace triggers, three arguments are appended to *`cmd`* so that the actual command is as follows:

```
cmd name1 name2 op
```

`name1` and *`name2`* give the name(s) for the variable being accessed: if the variable is a scalar then *`name1`* gives the variable's name and *`name2`* is an empty string; if the variable is an array element then *`name1`* gives the name of the array and *`name2`* gives the index into the array; if an entire array is being deleted and the trace was registered on the overall array, rather than a single element, then *`name1`* gives the array name and *`name2`* is an empty string. *`op`* indicates what operation is being performed on the variable, and is one of '`r`', '`w`', or '`u`' as defined above.

`cmd` executes in the same context as the code that invoked the traced operation: if the variable was accessed as part of a Tcl procedure, then *`cmd`* will have access to the same

local variables as code in the procedure. This context may be different than the context in which the trace was created. If *cmd* invokes a procedure (which it normally does) then the procedure will have to use `upvar` or `uplevel` if it wishes to access the traced variable. Note also that *name1* may not necessarily be the same as the name used to set the trace on the variable; differences can occur if the access is made through a variable defined with the `upvar` command.

For read and write traces, *cmd* can modify the variable to affect the result of the traced operation. If *cmd* modifies the value of a variable during a read or write trace, then the new value will be returned as the result of the traced operation. The return value from *cmd* is ignored except that if it returns an error of any sort then the traced operation also returns an error with the same error message returned by the `trace` command (this mechanism can be used to implement read-only variables, for example). For write traces, *cmd* is invoked after the variable's value has been changed; it can write a new value into the variable to override the original value specified in the write operation. To implement read-only variables, *cmd* will have to restore the old value of the variable.

While *cmd* is executing during a read or write trace, traces on the variable are temporarily disabled. This means that reads and writes invoked by *cmd* will occur directly, without invoking *cmd* (or any other traces) again. However, if *cmd* unsets the variable then unset traces will be invoked.

When an unset trace is invoked, the variable has already been deleted: it will appear to be undefined with no traces. If an unset occurs because of a procedure return, then the trace will be invoked in the variable context of the procedure being returned to: the stack frame of the returning procedure will no longer exist. Traces are not disabled during unset traces, so if an unset `trace` command creates a new trace and accesses the variable, the trace will be invoked. Any errors in unset traces are ignored.

If there are multiple traces on a variable they are invoked in order of creation, most-recent first. If one trace returns an error, then no further traces are invoked for the variable. If an array element has a trace set, and there is also a trace set on the array as a whole, the trace on the overall array is invoked before the one on the element.

Once created, the trace remains in effect either until the trace is removed with the '`trace vdelete`' command described below, until the variable is unset, or until the interpreter is deleted. Unsetting an element of array will remove any traces on that element, but will not remove traces on the overall array.

This command returns an empty string.

`trace vdelete` *`name ops cmd`*
: If there is a trace set on variable *`name`* with the operations and command given by *`ops`* and *`cmd`*, then the trace is removed, so that *`cmd`* will never again be invoked. Returns an empty string.

`trace vinfo` *`name`*
: Returns a list containing one element for each trace currently set on variable *`name`*. Each element of the list is itself a list containing two elements, which are the ops and command associated with the trace. If *`name`* doesn't exist or doesn't have any traces set, then the result of the command will be an empty string.

B.3.58 Handle non-existent commands: `unknown`

`unknown` *`cmd`* `[` *`arg`* `... ]`

This command doesn't actually exist as part of Tcl, but Tcl will invoke it if it does exist. If the Tcl interpreter encounters a command name for which there is not a defined command, then Tcl checks for the existence of a command named `unknown`. If there is no such command, then the interpreter returns an error. If the `unknown` command exists, then it is invoked with arguments consisting of the fully-substituted name and arguments for the original non-existent command. The `unknown` command typically does things like searching through library directories for a command procedure with the name *`cmd`*, or expanding abbreviated command names to full-length, or automatically executing unknown commands as sub-processes. In some cases (such as expanding abbreviations) `unknown` will change the original command slightly and then (re-)execute it. The result of the `unknown` command is used as the result for the original non-existent command.

B.3.59 Delete variables: `unset`

`unset` *`varname`* `[` *`varname`* `... ]`

This command removes one or more variables. Each *`varname`* is a variable name, specified in any of the ways acceptable to the `set` command. If a *`varname`* refers to an element of an array then that element is removed without affecting the rest of the array. If a *`varname`* consists of an array name with no parenthesized index, then the entire array is deleted. The `unset` command returns an empty string as result. An error occurs if any of the variables doesn't exist, and any variables after the non-existent one are not deleted.

B.3.60 Execute in another stack frame: `uplevel`

`uplevel [` *`lev`* `]` *`arg`* `[` *`arg`* `... ]`

All of the *`arg`* arguments are concatenated as if they had been passed to `concat`; the result is then evaluated in the variable context indicated by *`lev`*. `uplevel` returns the result of that evaluation.

If *`lev`* is an integer then it gives a distance (up the procedure calling stack) to move before executing the command. If *`lev`* consists of '`#`' followed by a number then the number gives an absolute level number. If *`lev`* is omitted then it defaults to 1. *`lev`* cannot be defaulted if the first command argument starts with a digit or '`#`'.

For example, suppose that procedure `a` was invoked from top-level, and that it called `b`, and that `b` called `c`. Suppose that `c` invokes the `uplevel` command. If *`lev`* is `1` or `#2` or omitted, then the command will be executed in the variable context of `b`. If *`lev`* is `2` or `#1` then the command will be executed in the variable context of `a`. If *`lev`* is `3` or `#0` then the command will be executed at top-level (only global variables will be visible).

The `uplevel` command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose `c` invokes the command

```
uplevel 1 {set x 43; d}
```

where `d` is another Tcl procedure. The `set` command will modify the variable `x` in the context of `b`, and `d` will execute at level 3, as if called from `b`. If it in turn executes the command

```
uplevel {set x 42}
```

then the `set` command will modify the same variable `x` in the context of `b`: the procedure `c` does not appear to be on the call stack when `d` is executing. The command '`info level`' may be used to obtain the level of the current procedure.

`uplevel` makes it possible to implement new control constructs as Tcl procedures (for example, `uplevel` could be used to implement the `while` construct as a Tcl procedure).

B.3.61 Link to different stack frame: `upvar`

`upvar [ `*`lev`*` ] `*`other mine`*` [ `*`other mine`*` . . . ]`

This command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call or to global variables. *`lev`* may have any of the forms permitted for the `uplevel` command, and may be omitted if the first letter of the first *`other`* isn't '#' or a digit (it defaults to 1). For each *`other`* argument, `upvar` makes the variable by that name in the procedure frame given by *`lev`* (or at global level, if *`lev`* is #0) accessible in the current procedure by the name given in the corresponding *`mine`* argument. The variable named by *`other`* need not exist at the time of the call; it will be created the first time *`mine`* is referenced, just like an ordinary variable. `upvar` may only be invoked from within procedures. *`mine`* may not refer to an element of an array, but *`other`* may refer to an array element. `upvar` returns an empty string.

The `upvar` command simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as Tcl procedures. For example, consider the following procedure:

```
proc add2 name {
     upvar $name x
     set x [expr $x+2]
}
```

`add2` is invoked with an argument giving the name of a variable, and it adds two to the value of that variable. Although `add2` could have been implemented using `uplevel` instead of `upvar`, `upvar` makes it simpler for `add2` to access the variable in the caller's procedure frame.

If an `upvar` variable is unset (e.g. `x` in `add2` above), the unset operation affects the variable it is linked to, not the `upvar` variable. There is no way to unset an `upvar` variable except by exiting the procedure in which it is defined. However, it is possible to retarget an `upvar` variable by executing another `upvar` command.

B.3.62 Execute script repeatedly: `while`

`while` *`tst body`*

The `while` command evaluates *`tst`* as an expression (see Section B.2 “Writing expressionf in Tcl,” page 90). The value of the expression must a proper Boolean value; if it is a true value then *`body`* is executed by passing it to the Tcl interpreter. Once *`body`* has been executed then *`tst`* is evaluated again, and the process repeats until eventually *`tst`* evaluates to a false Boolean value. `continue` commands may be executed inside *`body`* to terminate the current iteration of the loop, and `break` commands may be executed inside *`body`* to cause immediate termination of the `while` command. The `while` command always returns an empty string.

B.4 Variables built into Tcl

The following global variables are created and managed automatically by Tcl. Except where noted below, these variables should normally be treated as read-only by application-specific code and by users.

`env`
: This variable is maintained by Tcl as an array whose elements are the environment variables for the process. Reading an element will return the value of the corresponding environment variable. Setting an element of the array will modify the corresponding environment variable or create a new one if it doesn't already exist. Unsetting an element of `env` will remove the corresponding environment variable. Changes to the `env` array will affect the environment passed to children by commands like `exec`. If the entire `env` array is unset then Tcl will stop monitoring `env` accesses and will not update environment variables.

`errorCode`
: After an error has occurred, this variable will be set to hold additional information about the error in a form that is easy to process with programs. `errorCode` consists of a Tcl list with one or more elements. The first element of the list identifies a general class of errors, and determines the format of the rest of the list. The following formats for `errorCode` are used by the Tcl core; individual applications may define additional formats.

  `ARITH` *`code msg`*
  : This format is used when an arithmetic error occurs (e.g. an attempt to divide by zero in the `expr` command). *`code`* identifies the precise error and *`msg`* provides a human-readable description of the

error. *code* will be either `DIVZERO` (for an attempt to divide by zero), `DOMAIN` (if an argument is outside the domain of a function, such as '`acos(-3)`'), `IOVERFLOW` (for integer overflow), `OVERLFLOW` (for a floating-point overflow), or `UNKNOWN` (if the cause of the error cannot be determined).

`CHILDKILLED` *pid signame msg*
: This format is used when a child process has been killed because of a signal. The second element of `errorCode` will be the process identifier (in decimal). The third element will be the symbolic name of the signal that caused the process to terminate; it will be one of the names from the include file '`signal.h`', such as `SIGPIPE`. The fourth element will be a short human-readable message describing the signal, such as '`write on pipe with no readers`' for `SIGPIPE`.

`CHILDSTATUS` *pid code*
: This format is used when a child process has exited with a non-zero exit status. The second element of `errorCode` will be the process identifier (in decimal) and the third element will be the exit code returned by the process (also in decimal).

`CHILDSUSP` *pid signame msg*
: This format is used when a child process has been suspended because of a signal. The second element of `errorCode` will be the process identifier, in decimal. The third element will be the symbolic name of the signal that caused the process to suspend; this will be one of the names from the include file '`signal.h`', such as `SIGTTIN`. The fourth element will be a short human-readable message describing the signal, such as '`background tty read`' for `SIGTTIN`.

`NONE`
: This format is used for errors where no additional information is available for an error besides the message returned with the error. In these cases `errorCode` will consist of a list containing a single element whose contents are `NONE`.

`POSIX` *errname msg*

If the first element of `errorCode` is `POSIX`, then the error occurred during a POSIX kernel call. The second element of the list will contain the symbolic name of the error that occurred, such as `ENOENT`; this will be one of the values defined in the include file '`errno.h`'. The third element of the list will be a human-readable message corresponding to *errname*, such as '`no such file or directory`' for the `ENOENT` case.

To set `errorCode`, applications may invoke the `error` command (or an equivalent Tcl library subroutine). Otherwise the Tcl interpreter will reset the variable to `NONE` after the next error.

`errorInfo`

After an error has occurred, this string will contain one or more lines identifying the Tcl commands and procedures that were being executed when the most recent error occurred. Its contents take the form of a stack trace showing the various nested Tcl commands that had been invoked at the time of the error.

`tcl_precision`

If this variable is set, it must contain a decimal number giving the number of significant digits to include when converting floating-point values to strings. If this variable is not set then 6 digits are included. 17 digits is "perfect" for IEEE floating-point in that it allows double-precision values to be converted to strings and back to binary with no loss of precision.

B.5 Tcl standard library

Tcl includes a library of Tcl procedures for commonly-needed functions. The procedures defined in the Tcl library are generic ones suitable for use by many different applications. The location of the Tcl library is returned by the '`info library`' command. In addition to the Tcl library, each application will normally have its own library of support procedures as well; the location of this library is normally given by the value of the *app*`_library` global variable, where *app* is the name of the application. For example, the location of the Tk library is kept in the variable `tk_library`.

To access the procedures in the Tcl library, an application should run the file '`init.tcl`' in the library, for example with the Tcl command

```
source [info library]/init.tcl
```

This will define the `unknown` procedure and arrange for the other procedures to be loaded on demand using the auto-load mechanism defined below.

B.5.1 Procedures

The following procedures are provided in the Tcl library:

`auto_execok` *cmd*
: Determines whether there is an executable file by the name *cmd*. This command examines the directories in the current search path (given by the `PATH` environment variable) to see if there is an executable file named *cmd* in any of those directories. If so, it returns `1`; if not it returns `0`. `auto_execok` remembers information about previous searches in an array named `auto_execs`; this avoids the path search in future calls for the same *cmd*. The command `auto_reset` may be used to force `auto_execok` to forget its cached information.

`auto_load` *cmd*
: This command attempts to load the definition for a Tcl command named *cmd*. To do this, it searches an autoload path, which is a list of one or more directories. The auto-load path is given by the global variable `auto_path` if it exists. If there is no `auto_path` variable, then the `TCLLIBPATH` environment variable is used, if it exists. Otherwise the auto-load path consists of just the Tcl library directory. Within each directory in the auto-load path there must be a file '`tclIndex`' that describes one or more commands defined in that directory and a script to evaluate to load each of the commands. The '`tclIndex`' file should be generated with the `auto_mkindex` command. If *cmd* is found in an index file, then the appropriate script is evaluated to create the command. The `auto_load` command returns `1` if *cmd* was successfully created. The command returns `0` if there was no index entry for *cmd* or if the script didn't actually define *cmd* (e.g. because index information is out of date). If an error occurs while processing the script, then that error is returned. `auto_load` only reads the index information once and saves it in the array `auto_index`; future calls to `auto_load` check for *cmd* in the array rather than rereading the index files. The cached index information may be deleted with the command `auto_reset`. This will force the next `auto_load` command to reload the index database from disk.

`auto_mkindex` *dir ptn* [*ptn* . . .]

Generates an index suitable for use by `auto_load`. The command searches *dir* for all files whose names match any of the *ptn* arguments (matching is done with the `glob` command), generates an index of all the Tcl command procedures defined in all the matching files, and stores the index information in a file named '`tclIndex`' in *dir*. For example, the command

```
auto_mkindex foo *.tcl
```

will read all the '`.tcl`' files in subdirectory '`foo`' and generate a new index file '`foo/tclIndex`'.

`auto_mkindex` parses the Tcl scripts in a relatively unsophisticated way: if any line contains the word `proc` as its first characters then it is assumed to be a procedure definition and the next word of the line is taken as the procedure's name. Procedure definitions that don't appear in this way (e.g. they have spaces before the `proc`) will not be indexed.

`auto_reset`

Destroys all the information cached by `auto_execok` and `auto_load`. This information will be re-read from disk the next time it is needed. `auto_reset` also deletes any procedures listed in the auto-load index, so that fresh copies of them will be loaded the next time that they're used.

`parray` *arrayname*

Prints on standard output the names and values of all the elements in the array *arrayname*. *arrayname* must be an array accessible to the caller of `parray`. It may be either local or global.

`unknown` *cmd* [*arg* . . .]

This procedure is invoked automatically by the Tcl interpreter whenever the name of a command doesn't exist. The unknown procedure receives as its arguments the name and arguments of the missing command. Unknown first calls `auto_load` to load the command. If this succeeds, then it executes the original command with its original arguments. If the auto-load fails then unknown calls `auto_execok` to see if there is an executable file by the name *cmd*. If so, it invokes the Tcl `exec` command with *cmd* and all the instances of *arg* as arguments. If *cmd* can't be auto-executed, `unknown` checks to see if the command was invoked at top-level and outside of any script. If so, then `unknown` takes takes two additional steps. First, it sees if *cmd* has one of the following three forms: '`!!`', '`!`*event*', or '`^`*old*`^`*new*[`^`]'. If so, then `unknown` carries out history substitution in the same way that `csh` would for these constructs. Second, and last, `unknown` checks to see if *cmd* is a unique abbreviation for an existing Tcl command. If so, it expands the command name and executes the command with the original arguments. If none of the above efforts is able to execute the command, `unknown` generates an error return. If the global variable `auto_noload` is defined, then the auto-load step is skipped. If the global variable `auto_noexec` is defined then the auto-exec step is skipped. Under normal circumstances the return value from `unknown` is the return value from the command that was eventually executed.

B.5.2 Variables

The following global variables are defined or used by the procedures in the Tcl library:

`auto_execs`
: Used by `auto_execok` to record information about whether particular commands exist as executable files.

`auto_index`
: Used by `auto_load` to save the index information read from disk.

`auto_noexec`
: If set to any value, then `unknown` will not attempt to auto-exec any commands.

`auto_noload`
: If set to any value, then `unknown` will not attempt to auto-load any commands.

`auto_path`
: If set, then it must contain a valid Tcl list giving directories to search during auto-load operations.

`env(TCL_LIBRARY)`
: If set, then it specifies the location of the directory containing library scripts (the value of this variable will be returned by the command info library). If this variable isn't set then a default value is used.

`env(TCLLIBPATH)`
: If set, then it must contain a valid Tcl list giving directories to search during auto-load operations. This variable is only used if `auto_path` is not defined.

`unknown_active`
: This variable is set by `unknown` to indicate that it is active. It is used to detect errors where `unknown` recurses on itself infinitely. The variable is unset before `unknown` returns.

Appendix C Installing DejaGnu

Once you have the DejaGnu source unpacked and available, you must first configure the software to specify where it is to run (and the associated defaults); then you can proceed to installing it.

C.1 Configuring the DejaGnu test driver

Configuring DejaGnu records information about the host (where `runtest` executes) and target (where the tools and their output are tested). It is usually best to configure in a directory separate from the source tree, specifying where to find the source with the '`--srcdir`' option to `configure`:

```
configure host-type --target=target-type --srcdir=path
```

To run tests on a full-fledged operating system, you can specify host and target as the same machine; we call this a *native* configuration. For native configurations, specifying the target explicitly is not necessary. For example, to configure for a Sun-4 running SunOS:

```
configure sun4 --srcdir=/usr/cygnus/src/dejagnu
```

When the host and target are not the same (typically when testing software tools on a "bare board" controlled from a separate general-purpose computer), we refer to a *cross* configuration. For example, to run tests controlled by Sun-4 on an AMD 29000 board, communicating via the AMD "UDI" protocol:

```
configure sun4 --target=a29k-amd-udi \
    --srcdir=/usr/cygnus/src/dejagnu
```

You may also want to use the `configure` option '`--prefix`' to specify where you want DejaGnu and its supporting code installed. By default, installation is in subdirectories of '`/usr/local`', but you can select any alternate directory *`altdir`* by including '`--prefix=`*`altdir`*' on the `configure` command line. (This value is captured in the Makefile variables `prefix` and `exec_prefix`.)

DejaGnu uses Cygnus Configure. See section "What Configure Does" in *Cygnus Configure*, for more information on its capabilities, conventions, and command-line options.

Aside from the usual effect of preparing and building the executables on your host, configuring and making DejaGnu creates a file called '`site.exp`'. This file encapsulates information from both `configure` and `make`, as default values for DejaGnu tests. See Chapter 3 "Setting `runtest` defaults," page 15.

Save for a small number of example tests, the DejaGnu distribution itself does not include any test suites; these are available separately. Test suites for the GNU compiler (testing both GCC and G++) and for the GNU binary utilities are distributed in parallel with the DejaGnu distribution (but packaged as separate files). The test suite for the GNU debugger is distributed in parallel with each release of GDB itself, starting with GDB 4.9. After configuring the top-level DejaGnu directory, unpack and configure the test directories for the tools you want to test; then, in each test directory, run `make` to build auxiliary programs required by some of the tests.

C.2 Installing DejaGnu

To install DejaGnu in your filesystem (either in '`/usr/local`', or as specified by your '`--prefix`' option to `configure`), execute

```
eg$ make install
```

'`make install`' does these things for DejaGnu:

1. Look in the path specified for executables ('`$exec_prefix`') for directories called '`lib`' and '`bin`'. If these directories do not exist, '`make install`' creates them.
2. Create another directory in the '`lib`' directory, called '`dejagnu`'.
3. Create a subdirectory of '`lib/dejagnu`' whose name is the target configuration string. For a native configuration, '`make install`' creates two configuration-specific directories: one with the usual configuration name, and another called '`native`'.
4. Copy the `runtest` shell script into '`$exec_prefix/bin`'.
5. Copy all the library files (used to support the framework) into '`$exec_prefix/lib/dejagnu`'.
6. Copy '`runtest.exp`' into '`$exec_prefix/lib/dejagnu`'. This is the main Tcl code implementing DejaGnu.
7. Copy the '`site.tmpl`' file (which captures configuration values in Tcl variables) into a the directory '`$exec_prefix/lib/dejagnu/`'. '`site.tmpl`' is a template of a working config file. The installation process edits it into '`site.exp`'. Each site where DejaGnu is installed should edit the values in this file to reflect local practice.

Each test suite collection comes with simple installation instructions in a '`README`' file; in general, the test suites are designed to be unpacked in the source directory for the corresponding tool, and extract into a directory called '`testsuite`'.

Index

B

C

F

N

O

P

R

S

T

U