o

The DejaGnu Testing Framework

for DejaGnu Version 1.3

Jan 1996

Rob Savoye

Cygnus Support

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's output is designed to be parsed by other filtering script, and it is also human readable.

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`'.)

2 What is new in this release ?

This release has a number of substantial changes over version 1.2. The most visible change is that the version of expect and Tcl included in the release are up-to-date with the current stable net releases. Other changes are:

1. The config sub-system in DejaGnu has been completely redesigned. It now supports testing on remote hosts as well as remote targets.
2. More builtin support for building target binaries with the correct linker flags. Currently this only works with GCC, preferably with a target support by `libgloss`.
3. Lots of little bug fixes from a year of heavy use here at Cygnus Support.
4. DejaGnu now uses `autoconf` for configuration.
5. New test cases for DejaGnu have been added for the new features, plus the "–tool" option bug in the 1.2 testsuite has been fixed.
6. The `--tool` option is now optional.
7. `runtest` when searching for test drivers ignores all directories named SCCS, RCS, and CVS.
8. There is now a generic keyword based test harness that uses comments in source code to control how each test case gets built and run.
9. There is now some support for running a testsuite with multiple passes.

2.1 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 4.14.4 -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 Chapter 4 [Setting defaults for `runtest` options], page 15.

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 3 [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 output files '`gdb.sum`' and '`gdb.log`'.) See Section 5.7 [The files DejaGnu writes], page 35.

---

[1] '`--tool`' selects a particular suite of tests, *not* the name of the executable program to run. See Chapter 4 [Configuration dependent values], page 15, for information on the variables that you can use to specify the names of programs to run.

2.2 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.

2.3 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.

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

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...

2.4 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 assertions were 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. That is, it demonstrated that the assertion is true.

`XFAIL`
: POSIX 1003.3 does not incorporate the notion of expected failures, so `PASS`, instead of `XPASS`, must also be returned for test cases which were expected to fail and did not. This means that `PASS` is in some sense more ambiguous than if `XPASS` is also used. For information on `XPASS` and `XFAIL`, see Chapter 3 [Using `runtest`], page 9.

---

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

`FAIL` A test *has* produced the bug it was intended to capture. That is, it has demonstrated that the assertion is false. The `FAIL` message is based on the test case only. Other messages are used to indicate a failure of the framework.

As with `PASS`, POSIX tests must return `FAIL` rather than `XFAIL` even if a failure was expected.

`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 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.

Note that for POSIX, each assertion must produce a test result code. If the test isn't actually run, it must produce `UNRESOLVED` rather than just leaving that test out of the output. This means that you have to be careful when writing tests, to not carelessly use tcl statements like `return`—if you alter the flow of control of the tcl code you must insure that every test still produces some result code.

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. For a DejaGnu test suite to conform to POSIX 1003.3, you must avoid the `setup_xfail` procedure as described in the `PASS` section above, and you must be careful to return `UNRESOLVED` where appropriate, as described in the `UNRESOLVED` section above.

2.5 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.

2.6 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. The version of Tcl included in DejaGnu at this time is Tcl 7.4p3.

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`). The version of expect included in DejaGnu at this time is expect 5.18.0.

3 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.

`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 2.4 [A POSIX conforming test framework], page 6, 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.

`NOTE`
: An informational message about the test case.

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 --build 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 Chapter 4 [Configuration dependent values], page 15), 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 Chapter 4 [Configuration dependent values], page 15.

`--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 5.5 [Remote targets supported], page 32). 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 unless – build is also specified; 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`'. In this usage, `host` refers to the machine that the tests are to be run on, which may not be the same as the `build` machine. If `--build` is also specified, then `--host` refers to the machine that the tests wil, be run on, not the machine DejaGnu is run on.

`--build` *string*
: *string* is a full configuration "triple" name as used by `configure`. This is the type of machine DejaGnu and the tools to be tested are built on. For a normal cross this is the same as the host, but for a canadian cross, they are seperate.

`--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 5.5 [Remote targets supported], page 32.

`--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.

---

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

`--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⟩).

---

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

4 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). The first file loaded is the optional ~/.dejagnurc, then the local files, and finally the global file.

1. There is am optional "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. See Appendix A [Configuring and Installing DejaGnu], page 49. The master 'site.exp' contains the default values for all targets and hosts supported by DejaGnu. This master file is identified by setting the environment variable DEJAGNU to the name of the file. This is also refered to as the "global" config file.
2. 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 testsuite 'site.exp' by typing 'make site.exp' in the test suite directory, after the test suite is configured.
3. You can also have a file in your home directory called .dejagnurc. This gets loaded first before the other config files. Usually this is used for personal stuff, like setting all_flag so all the output gets printed, or verbosity levels.

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.

4.0.1 Config Variables

DejaGnu uses a named array in Tcl to hold all the info for each machine. In the case of a canadian cross, this means host information as well as target information. The named array is called target_info, and it has two indices. The following fields are part of the array.

name
: The name of the target. (mostly for error messages) This should also be the string used for this target's array. It should also be the same as the linker script so we can find them dynamically. This should be the same as the argument used for push_target{}.

ldflags
: This is the linker flags required to produce a fully linked executable. For libgloss supported targets this is usually just the name of the linker script.

config
: The target canonical for this target. This is used by some init files to make sure the target is supported.

cflags
: The flags required to produce an object file from a source file.

connect
: This is the connectmode for this target. This is for both IP and serial connections. Typically this is either telnet, rlogin, or rsh.

`target` This is the hostname of the target. This is for TCP/IP based connections, and is also used for version of tip that use /etc/remote.

`serial` This is the serial port. This is typically /dev/tty? or com?:.

`netport` This is the IP port. This is commonly used for telneting to target boards that are connected to a terminal server. In that case the IP port specifies the which serial port to use.

`baud` This is the baud rate for a serial port connection.

`x10` This is the parameters for an x10 controller. These are simple devices that let us power cycle or reset a target board remotely.

`fileid` This is the fileid or spawn id of of the connection.

`prompt` a glob style pattern to recognize the prompt.

`abbrev` abbreviation for tool init files.

`ioport` This is the port for I/O on dual port systems. In this configuration, the main serial port `0` is usually used for stdin and stdout, which the second serial port can be used for debugging.

The first index into the array is the same value as used in the `name` field. This is usually a short version of the name of the target board. For an example, here's the settings I use for my `Motorola's IDP` board and my `Motorola` 6U VME `MVME135-1` board. (both m68k targets)

```
# IDP board
set target_info(idp,name)        "idp"
set target_info(idp,ldflags)     "-Tidp.ld"
set target_info(idp,config)      m68k-unknown-aout
set target_info(idp,cflags)      ""
set target_info(idp,connect)     telnet
set target_info(idp,target)      "s7"
set target_info(idp,serial)      "tstty7"
set target_info(idp,netport)     "wharfrat:1007"
set target_info(idp,baud)        "9600"
# MVME 135 board
set target_info(idp,name)        "mvme"
set target_info(idp,ldflags)     "-Tmvme.ld"
set target_info(idp,config)      m68k-unknown-aout
set target_info(idp,cflags)      ""
set target_info(idp,connect)     telnet
set target_info(idp,target)      "s8"
set target_info(idp,serial)      "tstty8"
set target_info(idp,netport)     "wharfrat:1008"
set target_info(idp,baud)        "9600"
```

DejaGnu can use this information to switch between multiple targets in one test run. This is done through the use of the `push_target` procedure, which is discussed elsewhere.

This array can also hold information for a remote host, which is used when testing a candain cross. In this case, the only thing different is the index is just `host`. Here's the settings I use to run tests on my NT machine while running DejaGnu on a Unix machine. (in this case a Linux box)

```
set target_info(host,name)      "nt-host"
set target_info(host,config)    "386-unknown-winnt"
set target_info(host,connect)   "telnet"
set target_info(host,target)    "ripple"
```

There is more info on how to use these variables in the sections on the config files. See Chapter 4 [Configuration Files], page 18.

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 3 [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
`--target`	`target_triplet`	The canonical configuration string for the target.
`--host`	`host_triplet`	The canonical configuration string for the host.
`--build`	`build_triplet`	The canonical configuration string for the build host.

4.0.2 Master Config File

The master config file is where all the target specific config variables get set for a whole site get set. The idea is that for a centralized testing lab where people have to share a target between multiple developers. There are settings for both remote targets and remote hosts. Here's an example of a Master Config File (also called the Global config file) for a *canadian cross*. A canadian cross is when you build and test a cross compiler on a machine other than the one it's to be hosted on.

Here we have the config settings for our California office. Note that all config values are site dependant. Here we have two sets of values that we use for testing m68k-aout cross compilers. As both of these target boards has a different debugging protocol, we test on both of them in sequence.

```
global CFLAGS
global CXXFLAGS

case "$target_triplet" in {
    { "native" } {
        set target_abbrev unix
    }
    { "m68*-unknown-aout" } {
        set target_abbrev               "rom68k"
        # IDP target                    # IDP board with rom68k monitor
        set target_info(idp,name)       "idp"
        set target_info(idp,ldflags)    "-Tidp.ld"
        set target_info(idp,config)     m68k-unknown-aout
        set target_info(idp,cflags)     ""
        set target_info(idp,connect)    telnet
        set target_info(idp,target)     "s7"
        set target_info(idp,serial)     "tstty12"
        set target_info(idp,netport)    "truckin:1007"
        set target_info(idp,baud)       "9600"
        # MVME target                   # Motorola MVME 135 with BUG monitor
        set target_info(mvme,name)      "mvme"
        set target_info(mvme,ldflags)   "-Tmvme.ld"
        set target_info(mvme,config)    m68k-unknown-aout
        set target_info(mvme,cflags)    ""
        set target_info(mvme,connect)   telnet
        set target_info(mvme,target)    "s4"
        set target_info(mvme,serial)    "tstty8"
        set target_info(mvme,netport)   "truckin:1004"
        set target_info(mvme,baud)      "9600"
     }
}
```

In this case, we have support for several remote hosts for our m68k-aout cross compiler. Typically the remote Unix hosts run DejaGnu locally, but we also use them for debugging the testsuites when we find problems in running on remote hosts. Expect won't run on NT, so DejaGnu is run on the local build machine, and it'll connect to the NT host and run all the tests for this cross compiler on that host.

```
case "$host_triplet" in {
    "native" {
    }
    "i?86-*-linux*" {                   # Linux host
        set target_info(host,name)      "linux-host"
        set target_info(host,config)    $host_triplet
        set target_info(host,connect)   rlogin
        set target_info(host,target)    chinadoll
    }
    "i?86-*-winnt                       # NT host
        set target_info(host,name)      "nt-host"
        set target_info(host,config)    i386-unknown-winnt
        set target_info(host,connect)   telnet
        set target_info(host,target)    ripple
    }
    "hppa*-hp-hpux*" {                  # HP-UX host
        set target_info(host,name)      "hpux-host"
        set target_info(host,config)    $host_triplet
        set target_info(host,connect)   rlogin
        set target_info(host,target)    slipknot
        }
    "sparc-sun-sunos*" {                # SunOS (sun4)
        set target_info(host,name)      "sunos-host"
        set target_info(host,config)    $host_triplet
        set target_info(host,connect)   rlogin
        set target_info(host,target)    darkstar
    }
}
```

4.0.3 Local Config File

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.

All local ‘site.exp’ usually files have two sections, separated by comment text. 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. Do not edit this first section; if you do, your changes are replaced next time you run make.

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
```

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.)

The first section ends with this line:

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

You can make any changes under this line. If you wish to redefine a variable in the top section, then just put a duplicate value in this second section. Usually the values defined in this config file are related to the configuration of the test run. This is the ideal place to set the variables `host_triplet`, `build_triplet`, `target_triplet`. All other variables are tool dependant. ie for testing a compiler, the value for *CC* might be set to a freshly built binary, as opposed to one in the user's path.

4.0.4 Personal Config File

The personal config file is used to customize `runtest's` behaviour for each person. It's typically used to set the user prefered setting for verbosity, and any experimental Tcl procedures. My personal '`~/.dejagnurc`' file looks like:

```
set all_flag 1
set RLOGIN /usr/ucb/rlogin
set RSH /usr/ucb/rsh
```

Here I set `all_flag` so I see all the test cases that PASS along with the ones that FAIL. I also set *RLOGIN* and `RSH` to the BSD version. I have `kerberos` installed, and when I rlogin to a target board, it usually isn't supported. So I use the non secure versions of these programs rather than the default that's in my path.

5 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.

5.1 Conventions for using tool names

DejaGnu uses '`$tool`', the name of the tool under test, to tie together the testing configuration in a straightforward but flexible way. If there is only one testsuite for a particular application, then '`$tool`' is optional.

'`$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. It is a common practice to put a period after the tool part of the name. 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.

5.2 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 can be named several ways. The prefered name is the *os* part of the canonical configuration name with `.exp` as the suffix. An example would be that for an `m68k-coff` system, the `target_os` part would be `coff`. The next way is for system where there are short filenames, or a shortcut is desired to refer to the OS name for that target. This is uses the value of `$target_abbrev` rather than the `target_os`.

The final file looked for is simply '`default.exp`'. If there is only one operating system to support, then this file can be used. It's main purpose is to offer some support for new operating systems, or for unsupported cross targets. The last file looked for is '`unknown.exp`'. This is usually limited to error handling for unsupported targets. It's whole contents is typically.

```
perror "Sorry, there is no support for this target"
exit 1
```

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 AR
# look for the archiver ar
if ![info exists AR] {
    set AR [findfile $base_dir/../../binutils/ar $base_dir/../../binutils/ar [tr
ansform ar]]
    verbose "AR defaulting to $AR" 2
}
}

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 `AR` is set on the command line that invokes '`runtest`'.

The `findfile` procedure takes as it's first argument a file name to look for. The second argument is returned if the file is found, and the third argument is returned if the file is not found. `base_dir` is set internally by DejaGnu to the top level directory of the object tree.

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 5.4 [Target dependent procedures], page 31, for full descriptions of these procedures.

5.3 DejaGnu procedures

DejaGnu provides these Tcl procedures for use in test scripts. You can also use any standard `expect` or Tcl function. These procedures are stored in libraries, which DejaGnu loads at runtime. Here's explanation of the library procedures that get loaded at runtime. All other librarys are optional, and need to be loaded by the testsuite.

5.3.1 Core Internal Procedures

See Section 2.4 [A POSIX conforming test framework], page 6, for more detailed explanations of the test outcomes ('`FAIL`', '`PASS`', '`UNTESTED`', '`UNRESOLVED`', '`UNSUPPORTED`').

`perror "`*string number*`"`

Declares a severe error in the testing framework itself. `perror` writes in the log files a message beginning with '`ERROR`', appending the argument *string*. If the optional *number* is supplied, then this is used to set the internal count of errors to that value.

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. If the optional numeric value is '`0`', then there are no further side effects to calling this function, and the following test outcome doesn't become '`UNRESOLVED`'. This can be used for errors with no known side effects.

`warning "`*string number*`"`

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. If the optional *number* is supplied, then this is used to set the internal count of warnings to that value.

As a side effect, `warning_threshold` or more calls to `warning` in a single test case also changes 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. If the optional numeric value is '`0`', then there are no further side effects to calling this function, and the following test outcome doesn't become '`UNRESOLVED`'. This can be used for errors with no known side effects.

`note` "*string*"
: Appends an informational message to the log file. `note` writes in the log files a message beginning with '`NOTE`', appending the argument *string*. Use `note` sparingly. `verbose` should be used for most such messages, but in cases where a message is needed in the log file regardless of the verbosity level use `note`.

`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*.

`get_warning_threshold`
: Returns the current value of `warning_threshold`. The default value is 3.

`set_warning_threshold` *threshold*
: Sets the value of `warning_threshold`. A value of `0` disables it: calls to `warning` will not turn a '`PASS`' or '`FAIL`' into an '`UNRESOLVED`'.

`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). If it is passed a NULL string, then it returns the name of the build canonical configuration.

isbuild "*host*"
: Tests for a particular *build 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). If it is passed a NULL string, then it returns the name of the build canonical configuration.

: item is3way "*host*" Tests for a canadian cross. This is when the tests will be run on a remotly hosted cross compiler. If it is a canadian cross, then the result is 1; otherwise the result is 0.

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` [`-log`] [`-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 '`-log`' argument to cause *string* to always be added to the log file, even if it won't be printed. Use the optional '`-n`' argument to print *string* without a trailing newline. Use the optional '`--`' argument if *string* begins with "-".

5.3.2 Remote Communication Procedures

'`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`.

It use the value of the `connect` field in the `target_info` array (was `connectmode` as the type of connection to make. Current supported connection types are tip, kermit, telnet, rsh, rlogin, and netdata. If the `--reboot` option was used on the runtest command line, then the target is rebooted before the connection is made.

`remote_open` *type*

Remote Connection Procedure. This is passed *host* or *target*. Host or target refers to whether it is a connection to a remote target, or a remote host. This opens the connection to the desired target or host using the default values in the configuration system. It returns that `spawn_id` of the process that manages the connection. This value can be used in `expect` or `exp_send` statements, or passed to other procedures that need the connection process's id. This also sets the `fileid` field in the `target_info` array.

`remote_close` *shellid*
: *shellid* is value returned by a call to `remote_open`. This closes the connection to the target so resources can be used by others. This parameter can be left off if the `fileid` field in the `target_info` array is set.

`telnet` *hostname port*
`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. The optional *port* is used to specify the IP port number. The value of the `netport` field in the `target_info` array is used. (was `$netport`) This value has two parts, the hostname and the port number, seperated by a *:*. If `host` or `target` is used in the `hostname` field, than the config array is used for all information.

`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. The value of the `serial` field in the `target_info` array is used. (was `$serialport`) If `host` or `target` is used in the `port` field, than the config array is used for all information.

`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. The value of the `serial` field in the `target_info` array is used. (was `$serialport`) If `host` or `target` is used in the `port` field, than the config array is used for all information.

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*.)

5.3.3 Utility Procedures

'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*. If no directories match the pattern, then a NULL string is returned.

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. If no files match the pattern, then a NULL string is returned.

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'.)

diff *filename filename*

Compares the two files and returns a 1 if they match, or a 0 if they don't. If verbose is set, then it'll print the differences to the screen.

slay *name*

This look in the process tabel for *name* and send it a unix SIGINT, killing the process.

absolute *path*

This procedure takes the relative *path*, and converts it to an absolute path.

psource *filename*

This sources the file *filename*, and traps all errors. It also ignores all extraneous output. If there was an error it returns a 1, otherwise it returns a 0.

`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*.

`setenv` *var val*
: Sets the variable *var* to the value *val*.

`unsetenv` *var*
: Unsets the environment variable *var*

`getenv` *var*
: returns the value of *var* in the environment if it exists, otherwise it returns `NULL`.

`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.

5.3.4 Cross target procedure

'`lib/target.exp`' defines these utility procedures:

`push_target` *name*
: This makes the target named *name* be the current target connection. The value of *name* is an index into the `target_info` array and is set in the global config file.

`pop_target`
: This unsets the current target connection.

`list_targets`
: This lists all the supported targets for this architecture.

`push_host` *name*
: This makes the host named *name* be the current remote host connection. The value of *name* is an index into the `target_info` array and is set in the global config file.

`pop_host` This unsets the current host connection.

: This invokes the compiler as set by `CC` to compile the file *file*. The default options for many cross compilation targets are *guessed* by DejaGnu, and these options can be added to by passing in more parameters as arguments to `compile`. Optionally, this will also use the value of the `cflags` field in the target config array. If the host is not the same as the build machines, then then compiler is run on the remote host using `execute_anywhere`.

This produces an archive file. Any parameters passed to `archive` are used in addition to the default flags. Optionally, this will also use the value of the `arflags` field in the target config array. If the host is not the same as the build machines, then then archiver is run on the remote host using `execute_anywhere`.

This generates an index for the archive file for systems that aren't POSIX yet. Any parameters passed to `ranlib` are used in for the flags.

`execute_anywhere` *cmdline*
: This executes the *cmdline* on the proper host. This should be used as a replacement for the Tcl command `exec` as this version utilizes the target config info to execute this command on the build machine or a remote host. All config information for the remote host must be setup to have this command work. If this is a canadian cross, (where we test a cross compiler that runs on a different host then where DejaGnu is running) then a connection is made to the remote host and the command is executed there. It returns either *REMOTERROR* (for an error) or the output produced when the command was executed. This is used for running the tool to be tested, not a test case.

5.3.5 Debugging Procedures

'`lib/debugger.exp`' defines these utility procedures:

`dumpvars` *expr*
: This takes a csh style regular expression (glob rules) and prints the values of the global variable names that match. It is abbreviated as `dv`

`dumplocals` *expr*
: This takes a csh style regular expression (glob rules) and prints the values of the local variable names that match. It is abbreviated as `dl`.

`dumprocs` *expr*
: This takes a csh style regular expression (glob rules) and prints the body of all procs that match. It is abbreviated as `dp`

`dumpwatch` *expr*
: This takes a csh style regular expression (glob rules) and prints all the watchpoints. It is abbreviated as `dw`.

`watchunset` *var*
: This breaks program execution when the variable *var* is unset. It is abbreviated as `wu`.

`watchwrite` *var*
: This breaks program execution when the variable *var* is written. It is abbreviated as `ww`.

`watchread` *var*
: This breaks program execution when the variable *var* is read. It is abbreviated as `wr`.

`watchdel` *watch*
: This deletes a the watchpoint for *watch*. It is abbreviated as `wd`.

`print` *var*
: This prints the value of the variable *var*. It is abbreviated as `p`.

`quit`
: This makes runtest exit. It is abbreviated as `q`.

`bt`
: This prints a backtrace of the executed Tcl commands.

5.4 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 5.2 [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.

5.5 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.

5.6 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 4 [Setting `runtest` defaults], page 15, for details.
2. '`lib/utils.exp`', a collection of utility procedures. See Section 5.3 [DejaGnu Builtins], page 23, for descriptions of these procedures.
3. '`lib/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. '`lib/remote.exp`', a collection of subroutines meant for connecting to remote machines.
6. '`lib/target.exp`', a collection of subroutines used for the configuration systems in DejaGnu. These procedures typically manipulate or utilize the configuration system.
7. An initialization file *tool*`_init.exp`. See Section 5.2 [Initialization module], page 22, for more discussion of init files.

5.7 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.

5.7.1 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
```

5.7.2 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
```

5.7.3 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.

6 How To Write a Test Case

6.1 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.

6.2 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 5.7.3 [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 3 [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`'.

6.3 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*.

6.4 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 5.3 [DejaGnu Builtins], page 23.

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 5.7.3 [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.

6.5 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 4 [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.

7 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.

7.1 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.

7.2 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 5.2 [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 5.3 [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 5.3 [DejaGnu Builtins], page 23.

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.

7.3 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 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.

A.1 Configuring the DejaGnu test driver

It is usually best to configure in a directory separate from the source tree, specifying where to find the source with the optional '--srcdir' option to `configure`. DejaGnu uses the GNU `autoconf` to configure itself. For more info on using autoconf, read the GNU autoconf manual. To configure, execute the '`configure`' program, no other options are required. For an example, to configure in a seperate tree for objects, execute the configure script from the source tree like this:

```
../dejagnu-1.3/configure
```

DejaGnu doesn't care at config time if it's for testing a native system or a cross system. That is determined at runtime by using the config files.

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`.)

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.

A.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 ('`$exec_prefix`') for directories called '`lib`' and '`bin`'. If these directories do not exist, '`make install`' creates them.
2. Create another directory in the '`lib`' directory, called '`dejagnu`'.
3. Copy the `runtest` shell script into '`$exec_prefix/bin`'.
4. Copy all the library files (used to support the framework) into '`$exec_prefix/lib/dejagnu`'.
5. Copy '`runtest.exp`' into '`$exec_prefix/lib/dejagnu`'. This is the main Tcl code implementing DejaGnu.

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

-

.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Table of Contents