DejaGnu

1 Introduction

1.1 What is DejaGnu?

DejaGnu is a framework for testing other programs, providing a single front-end for all tests. You can think of it as a library of Tcl procedures to help with writing a test harness. A *test harness* is the infrastructure that is created to test a specific program or tool. Each program can have multiple testsuites, all supported by a single test harness. DejaGnu is written in Expect, which in turn uses Tcl, the Tool command language. There is more information on Tcl at the Tcl/Tk web site (`http://www.tcl.tk`) and the Expect web site (`http://expect.nist.gov`).

Julia Menapace first coined the term *DejaGnu* to describe an earlier testing framework she wrote at Cygnus Support for testing GDB. When we replaced it with the Expect-based framework, it was like DejaGnu all over again. More importantly, it was also named after my daughter, Deja Snow Savoye, who was a toddler during DejaGnu's beginnings.

DejaGnu offers several advantages for testing:

- The flexibility and consistency of the DejaGnu framework make it easy to write tests for any program, with either batch-oriented, or interactive programs.
- 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 supported host system on any supported target system. DejaGnu runs tests on many single board computers, whose operating software ranges from a simple boot monitor to a real-time OS.
- 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.
- Using Tcl and Expect, it's easy to create wrappers for existing testsuites. By incorporating existing tests under DejaGnu, it's easier to have a single set of report analyse programs..

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

1.2 New in this release

The following major, user-visible changes have been introduced since version 1.5.3.

1. Support for target communication via SSH has been added.
2. A large number of very old config and baseboard files have been removed. If you need to resurrect these, you can get them from version 1.5.3. If you can show that a board is still in use, it can be put back in the distribution.
3. The `--status` command line option is now the default. This means that any error in the testsuite Tcl scripts will cause runtest to abort with exit status code 2. The

`--status` option has been removed from the documentation, but will continue to be accepted for backward compatibility.

4. `runtest` now exits with exit code 0 if the testsuite "passed", 1 if something unexpected happened (eg, FAIL, XPASS or UNRESOLVED), and 2 if an exception is raised by the Tcl interpreter.
5. `runtest` now exits with the standard exit codes of programs that are terminated by the SIGINT, SIGTERM and SIGQUIT signals.
6. The user-visible utility procedures `absolute`, `psource` and `slay` have been removed. If a testsuite uses any of these procedures, a copy of the procedure should be made and placed in the lib directory of the testsuite.
7. Support was added for testing the D compiler.
8. `~/.dejagnurc` is now loaded last, not first. This allows the user to have the ability to override anything in their environment (even the `site.exp` file specified by `$DEJAGNU`).
9. The user-visible utility procedure `unsetenv` is **deprecated** and will be removed in the next release. If a testsuite uses this procedure, a copy should be made and placed in the lib directory of the testsuite.

1.3 Design goals

DejaGnu grew out of the internal needs of Cygnus Solutions (formerly Cygnus Support). Cygnus maintained and enhanced a variety of free programs in many different environments and needed a testing tool that:

- was useful to developers while fixing bugs;
- automated running many tests during a software release process;
- was portable among a variety of host computers;
- supported a cross-development environment;
- permitted testing of interactive programs like `GDB`; and
- permitted testing of batch-oriented programs like `GCC`.

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

Probably the greatest challenge was testing in a cross-development environment. 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 written, any test can use it. Currently DejaGnu can use `ssh`, `rsh`, `rlogin`, `telnet`, `tip`, and `kermit` for remote communications.

1.4 A POSIX compliant test framework

DejaGnu conforms to the POSIX 1003.3 standard for test frameworks. Rob Savoye was a member of that committee.

POSIX standard 1003.3 defines what a testing framework needs to provide to create a POSIX compliant testsuite. This standard is primarily oriented to checking POSIX conformance, 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 was created by Unisoft for a consortium comprised of X/Open, Unix International and the Open Software Foundation.

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 day or night. 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 testsuite to verify that testing frameworks are POSIX 1003.3 compliant, this is done by repeatedly reading the standard and experimenting. One of the main things POSIX 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 all five output messages. In this sense a testsuite that uses exactly these messages can be considered POSIX compliant. These definitions specify the output of a test case:

PASS
: A test has succeeded. That is, it demonstrated that the assertion is true.

FAIL
: A test has not succeeded – the assertion is false. The *FAIL* message is based on this 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.

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.

UNRESOLVED
: A test produced indeterminate results. Usually, this means the test executed in an unexpected fashion. This outcome requires a human to 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 commands 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*:

- Execution of a test 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 *WARN-*

ING messages. Any *WARNING* or *ERROR* messages can invalidate the output of the test. This usually requires a human to examine the output to determine what really happened – and to improve the test case.

- A test depends on a previous test, which has failed.
- 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.

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 testsuites and these procedures are already known to conform to POSIX 1003.3. For a DejaGnu testsuite 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.

1.5 Installation

Refer to the `INSTALL` in the source distribution for detailed installation instructions. Note that there is no compilation step as with many other GNU packages, as DejaGnu consists of interpreted code only.

Save for its own small testsuite, the DejaGnu distribution does not include any testsuites. Testsuites for the various GNU development tools are included with those packages. 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 check* to build auxiliary programs required by some of the tests, and run the test suites.

2 Running tests

There are two ways to execute a testsuite. The most common way is when there is existing support in the `Makefile` of the tool being tested. This usually consists of a *check* target. The other way is to execute the `runtest` program directly. To run `runtest` directly from the command line requires either all of the correct command line options, or a Section 3.2 [Local config file], page 16, must be set up correctly.

2.1 Running 'make check'

To run tests from an existing collection, first use `configure` as usual to set up the build directory. Then type `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. The most common file the *check* target depends on is the `site.exp` file. The `site.exp` contains various variables that DejaGnu uses to determine the configuration of the program being tested.

Once you have run *make check* to build any auxiliary files, you can invoke the test driver `runtest` directly to repeat the tests. You will also have to execute `runtest` directly for test collections with no *check* target in the `Makefile`.

GNU Automake has built-in support for DejaGnu. To add DejaGnu support to your generated `Makefile.in`, just add the keyword `dejagnu` to the AUTOMAKE_OPTIONS variable in `Makefile.am`. This will ensure that the generated `Makefile.in` has a `check` target that invokes DejaGnu correctly.

2.2 Running runtest

`runtest` is the test driver for DejaGnu. You can specify two kinds of things on the `runtest` command line: command line options, and Tcl variables that are passed to the test scripts. The options are listed alphabetically below.

`runtest` returns one of the following exit codes:

0 if all tests passed including expected failures and unsupported tests.

1 if any test failed, passed unexpectedly, or was unresolved.

2 if Expect encountered any error in the test scripts.

2.2.1 Output States

`runtest` flags the outcome of each test as one of these cases. See Section 1.4 [A POSIX Conforming Test Framework], page 3, for a discussion of how POSIX specifies the meanings of these cases.

PASS The most desirable outcome: the test was expected to succeed and did 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 testsuite 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 testsuites 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.

2.2.2 Invoking runtest

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

`-a`, `--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).

`--build [triplet]`
triplet is a system triplet of the form *cpu-vendor-os*. This is the type of machine DejaGnu and the tools to be tested are built on. For a normal cross environment this is the same as the host, but for a Canadian cross, they are different.

`-D0`, `-D1` Start the internal Tcl debugger. The Tcl debugger supports breakpoints, single stepping, and other common debugging activities. See the document Debugger for Tcl Applications (`http://expect.sourceforge.net/doc/tcl-debug.ps`)

by Don Libes. 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 `Controlc`).

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

`--host [triplet]`
triplet is a system triplet of the form *cpu-vendor-os*. 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 will be run on, not the machine DejaGnu is run on.

`--host_board [name]`
The host board to use.

`--ignore [name(s)]`
The name(s) of specific tests to ignore.

`--log_dialog`
Emit Expect output to stdout. The Expect output is usually only written to the `.log` file. By enabling this option, they are also printed to standard output.

`--mail [address(es)]`
Send test results to one or more email addresses.

`--objdir [path]`
Use *path* as the top directory containing any auxiliary compiled test code. The default is '.'. Use this option to locate pre-compiled test code. You can normally prepare any auxiliary files needed with *make*.

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

`--reboot [name]`
: Reboot the target board when `runtest` starts. 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 can take 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 [triplet]`
: Use this option to override the default setting (native testing). *triplet* is a system triplet of the form *cpu-vendor-os*. This option changes the configuration `runtest` uses for the default tool names, and other setup information.

`--target_board [name(s)]`
: The list of target boards to run tests on.

`--tool [name(s)]`
: Specifies which testsuite 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` and you can override them by specifying Tcl variables on the command line.

  For example, including `--tool` gcc on the 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=path-to-gcc* on the *runtest* command line.

`--tool_exec [name]`
: The path to the tool executable to test.

`--tool_opts [options]`
: A list of additional options to pass to the tool.

`-v`, `--verbose`
: 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.

`-V`, `--version`
: Prints out the version numbers of DejaGnu, Expect, and Tcl.

`-x`, `--xml=FILE`

Generate XML output. FILE is optional; if given it is the name of the output file. If not given, the output file is named after the tool.

`testfile`.exp[=arg(s)]

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 cannot include directory names, only plain filenames.

`arg(s)` specifies 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 *?*, ***, and *[chars]* are recognized.

`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.

2.2.3 Common Options

Typically, you don't need to use any command line options. The `--tool` option is only required when there is more than one testsuite in the same directory. The default options are in the local `site.exp` file, created by `make site.exp`.

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

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

The test output follows, then ends 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:

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

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 `-a` (all) option. For more verbose output about processes being run, communication, and so on, use `-v` (verbose). To see even more output, use multiple `-v` options. See Section 2.2.2 [Invoking runtest], page 7, for a more detailed explanation of each `runtest` option.

2.3 Output files

DejaGnu always writes two kinds of output files. Summary output is written to the `.sum` file, and detailed output is written to the `.log` file. The tool name determines the prefix for these files. For example, after running with `--tool gdb`, the output files will be called `gdb.sum` and `gdb.log`. For troubleshooting, a debug log file that logs the operation of Expect is available. Each of these will be described in turn.

2.3.1 Summary log file

DejaGnu always produces a summary (`.sum`) output file. This summary lists 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), the full pathname of the tool tested, and the version number of the tool. 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 running `runtest --tool binutils` a summary log file will be written to `binutils.sum`. Normally, DejaGnu writes this file in your current working directory. Use the `--outdir` option to select a different output directory.

Sample summary log

```
Test Run By bje on Sat Nov 14 21:04:30 AEDT 2015

 === 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
```

2.3.2 Detailed log file

DejaGnu also saves a detailed log file (`.log`), showing any output generated by test cases as well as the summary output. For example, after running `runtest --tool binutils`, a detailed log file will be written to `binutils.log`. Normally, DejaGnu writes this file in your current working directory. Use the `--outdir` option to select a different output directory.

Sample detailed log for g++ tests

```
Test Run By bje on Sat Nov 14 21:07:23 AEDT 2015

                === 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/latest/bin/g++ version cygnus-2.0.1
```

2.3.3 Debug log file

The `runtest` option `--debug` creates a file showing the output from Expect in debugging mode. The `dbg.log` file is created in the current directory. The log file shows the string sent to the tool being tested by each `send` command and the pattern it compares with the tool output by each `expect` command.

The log messages 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.

Example debug log file 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) }
epect: 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 standard output, and in the detailed log file.

3 Customizing DejaGnu

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. If this file is setup correctly, it is possible to execute a testsuite merely by typing `runtest`.

DejaGnu supports two `site.exp` files. The multiple instances of `site.exp` are loaded in a fixed order. The first file loaded is the local file `site.exp`, and then the optional global `site.exp` file as pointed to by the `DEJAGNU` environment variable.

There is an optional global `site.exp`, containing configuration values that apply to DejaGnu site-wide. `runtest` loads these values first. The global `site.exp` contains the default values for all targets and hosts supported by DejaGnu. This global file is identified by setting the environment variable `DEJAGNU` to the name of the file.

Any directory containing a configured testsuite also has a local `site.exp`, capturing configuration values specific to the tool being tested. 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 global `site.exp` file.

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

You can also have a file in your home directory called `.dejagnurc`. This gets loaded after the other config files. Usually this is used for personal stuff, like setting the `all_flag` so all the output gets printed, or your own verbosity levels. This file is usually restricted to setting command line options.

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.

3.1 Global config file

The global configuration file is where all the target specific configuration variables for a site are set. For example, a centralized testing lab where multiple developers have to share an embedded development board. There are settings for both remote hosts and remote targets. Below is an example of a global configuration file for a Canadian cross environment. A Canadian cross is a toolchain that is built on, runs on, and targets three different system triplets (for example, building a Solaris-hosted MIPS R4000 toolchain on a GNU/Linux system). All configuration values in the example below are site-specific.

Example global configuration file

```
# Make sure we look in the right place for the board description files.
lappend boards_dir "/nfs/cygint/s1/cygnus/dejagnu/boards"

verbose "Global config file: target_triplet is $target_triplet" 2
global target_list

case "$target_triplet" in {
    { "native" } {
        set target_list "unix"
```

```
    }
    { "sparc64-*elf" } {
        set target_list "sparc64-sim"
    }
    { "mips-*elf" } {
        set target_list "mips-sim wilma barney"
    }
    { "mips-lsi-elf" } {
        set target_list "mips-lsi-sim{,soft-float,el}"
    }
}
```

In this case, we have support for several cross compilers, that all run on this host. To run DejaGnu tests on tools hosted on operating systems that do not run Expect, DejaGnu can be run on the build machine and connect to the remote host to run all the tests. As you can see, all one does is set the variable `target_list` to the list of targets and options to test.

In this example, simple cases like *sparc64-elf* only require setting the name of the single board configuration file. The *mips-elf* target is more complicated and sets the list to three target boards. *mips-sim* is a symbolic name for a simulator "board" and *wilma* and *barney* are symbolic names for physical boards. Symbolic names are covered in the Section 4.4 [Adding a new board], page 25, section. The more complicated example is the entry for *mips-lsi-elf*. This one runs the tests with multiple iterations using all possible combinations of the `--soft-float` and the `--el` (little endian) options. The braced string includes an initial comma so that the set of combinations includes no options at all. Needless to say, this last target example is mostly specific to compiler testing.

3.2 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 global `site.exp` in the installed DejaGnu library. This file is mostly for supplying tool specific info that is required by the testsuite.

All local `site.exp` files have two sections, separated by comments. The first section 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`. 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 will be overwritten the 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 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 testsuite 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 dependent, i.e., 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.

Here's an example local site.exp file, as used for GCC/G++ testing.

Local Config File

```
## these variables are automatically generated by make ##
# Do not edit here. If you wish to override these values
# add them to the last section
set rootme "/build/devo-builds/i686-pc-linux-gnu/gcc"
set host_triplet i686-pc-linux-gnu
set build_triplet i686-pc-linux-gnu
set target_triplet i686-pc-linux-gnu
set target_alias i686-pc-linux-gnu
set CFLAGS ""
set CXXFLAGS "-isystem /build/devo-builds/i686-pc-linux-gnu/gcc/../libio -isystem $src
append LDFLAGS " -L/build/devo-builds/i686-pc-linux-gnu/gcc/../ld"
set tmpdir /build/devo-builds/i686-pc-linux-gnu/gcc/testsuite
set srcdir "${srcdir}/testsuite"
## All variables above are generated by configure. Do Not Edit ##
```

This file defines the required fields for a local config file, namely the three system triplets, and the srcdir. It also defines several other Tcl variables that are used exclusively by the GCC testsuite. For most test cases, the CXXFLAGS and LDFLAGS are supplied by DejaGnu itself for cross testing, but to test a compiler, GCC needs to manipulate these itself.

The local `site.exp` may also set Tcl variables such as `test_timeout` which can control the amount of time (in seconds) to wait for a remote test to complete. If not specified, `test_timeout` defaults to 300 seconds.

3.3 Board configuration file

The board configuration file is where board-specific configuration details are stored. A board configuration file contains all the higher-level configuration settings. There is a rough inheritance scheme, where it is possible to derive a new board description file from an existing one. There are also collections of custom procedures for common environments. For more information on adding a new board config file, go to the Section 4.4 [Adding a new board], page 25, section.

An example board configuration file for a GNU simulator is as follows. `set_board_info` is a procedure that sets the field name to the specified value. The procedures mentioned in brackets are *helper procedures*. These are used to find parts of a toolchain required to build an executable image that may reside in various locations. This is mostly of use when the startup code, the standard C libraries, or the toolchain itself is part of your build tree.

Example file

```
# This is a list of toolchains that are supported on this board.
set_board_info target_install {sparc64-elf}

# Load the generic configuration for this board. This will define any
# routines needed by the tool to communicate with the board.
load_generic_config "sim"

# We need this for find_gcc and *_include_flags/*_link_flags.
load_base_board_description "basic-sim"

# Use long64 by default.
process_multilib_options "long64"

setup_sim sparc64

# We only support newlib on this target. We assume that all multilib
# options have been specified before we get here.

set_board_info compiler "[find_gcc]"
set_board_info cflags "[libgloss_include_flags] [newlib_include_flags]"
set_board_info ldflags "[libgloss_link_flags] [newlib_link_flags]"
# No linker script.
set_board_info ldscript ""

# Used by a few gcc.c-torture testcases to delimit how large the
# stack can be.
set_board_info gcc,stack_size 16384
# The simulator doesn't return exit status and we need to indicate this
# the standard GCC wrapper will work with this target.
set_board_info needs_status_wrapper 1
# We can't pass arguments to programs.
set_board_info noargs 1
```

There are five helper procedures used in this example:

- `find_gcc` looks for a copy of the GNU compiler in your build tree, or it uses the one in your path. This will also return the proper transformed name for a cross compiler if you whole build tree is configured for one.
- `libgloss_include_flags` returns the flags to compile using Section 6.1.8 [Libgloss], page 57, the GNU board support package (BSP).
- `libgloss_link_flags` returns the flags to link an executable using Section 6.1.8 [Libgloss], page 57.
- `newlib_include_flags` returns the flags to compile using newlib (`https://sourceware.org/newlib`), a re-entrant standard C library for embedded systems comprising of non-GPL'd code
- `newlib_link_flags` returns the flags to link an executable with newlib (`https://`

sourceware.org/newlib).

3.4 Remote host testing

DejaGnu also supports running the tests on a remote host. To set this up, the remote host needs an FTP server, and a telnet server. Currently foreign operating systems used as remote hosts are VxWorks, VRTX, DOS/Windows 3.1, MacOS and Windows.

The recommended source for a Windows-based FTP server is to get IIS (either IIS 1 or Personal Web Server) from http://www.microsoft.com (`http://www.microsoft.com`). When you install it, make sure you install the FTP server - it's not selected by default. Go into the IIS manager and change the FTP server so that it does not allow anonymous FTP. Set the home directory to the root directory (i.e. c:\) of a suitable drive. Allow writing via FTP.

It will create an account like IUSR_FOOBAR where foobar is the name of your machine. Go into the user editor and give that account a password that you don't mind hanging around in the clear (i.e. not the same as your admin or personal passwords). Also, add it to all the various permission groups.

You'll also need a telnet server. For Windows, go to the Ataman (`http://ataman.com`) web site, pick up the Ataman Remote Logon Services for Windows, and install it. You can get started on the eval period anyway. Add IUSR_FOOBAR to the list of allowed users, set the HOME directory to be the same as the FTP default directory. Change the Mode prompt to simple.

Ok, now you need to pick a directory name to do all the testing in. For the sake of this example, we'll call it piggy (i.e. c:\piggy). Create this directory.

You'll need a unix machine. Create a directory for the scripts you'll need. For this example, we'll use /usr/local/swamp/testing. You'll need to have a source tree somewhere, say /usr/src/devo. Now, copy some files from releng's area in SV to your machine:

Remote host setup

```
cd /usr/local/swamp/testing
mkdir boards
scp darkstar.welcomehome.org:/dejagnu/cst/bin/MkTestDir .
scp darkstar.welcomehome.org:/dejagnu/site.exp .
scp darkstar.welcomehome.org:/dejagnu/boards/useless98r2.exp boards/foobar.exp
export DEJAGNU=/usr/local/swamp/testing/site.exp
```

You must edit the boards/foobar.exp file to reflect your machine; change the hostname (foobar.com), username (iusr_foobar), password, and ftp_directory (c:/piggy) to match what you selected.

Edit the global `site.exp` to reflect your boards directory:

Add The Board Directory

```
lappend boards_dir "/usr/local/swamp/testing/boards"
```

Now run MkTestDir, which is in the contrib directory. The first parameter is the toolchain prefix, the second is the location of your devo tree. If you are testing a cross compiler (ex: you have sh-hms-gcc.exe in your PATH on the PC), do something like this:

Setup Cross Remote Testing

```
./MkTestDir sh-hms /usr/dejagnu/src/devo
```

If you are testing a native PC compiler (ex: you have gcc.exe in your PATH on the PC), do this:

Setup Native Remote Testing

```
./MkTestDir '' /usr/dejagnu/src/devo
```

To test the setup, `ftp` to your PC using the username (iusr_foobar) and password you selected. CD to the test directory. Upload a file to the PC. Now telnet to your PC using the same username and password. CD to the test directory. Make sure the file is there. Type "set" and/or "gcc -v" (or sh-hms-gcc -v) and make sure the default PATH contains the installation you want to test.

Run Test Remotely

```
cd /usr/local/swamp/testing
make  -k -w check RUNTESTFLAGS="--host_board foobar --target_board foobar -v -v" > che
```

To run a specific test, use a command like this (for this example, you'd run this from the gcc directory that MkTestDir created):

Run a Test Remotely

```
make check RUNTESTFLAGS="--host_board sloth --target_board sloth -v compile.exp=921202
```

Note: if you are testing a cross-compiler, put in the correct target board. You'll also have to download more .exp files and modify them for your local configuration. The -v's are optional.

3.5 Config file values

DejaGnu uses a Tcl associative array 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.

3.5.1 Command line option variables

In the user editable second section of the Section 3.5.2 [User configuration file], page 21, you can not only override the configuration variables captured in the first section, but also specify default values for all on 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`. Section 2.2.2 [Invoking runtest], page 7, for explanations of the command-line options.

Option	Tcl variable	Description
-a, –all	all_flag	display all test results if set
–build	build_triplet	system triplet for the build host
–dir	cmdline_dir_to_run	run only tests in the specified directory

–host	host_triplet	system triplet for the host
–host_board	host_board	host board definition to use
–ignore	ignoretests	do not run the specified tests
–log_dialog	log_dialog	emit Expect output to standard output
–outdir	outdir	directory for `.sum` and `.log` files
–objdir	objdir	directory for pre-compiled binaries
–reboot	reboot	reboot the target if set to 1
–srcdir	srcdir	directory of test subdirectories
–target	target_triplet	system triplet for the target
–target_board	target_list	list of target boards to run tests on
–tool	tool	name of tool to test (identifies init, test subdirectory)
–tool_exec	TOOL_EXECUTABLE	path to the executable to test
–tool_opts	TOOL_OPTIONS	additional options to pass to the tool
–tool_root_dir	tool_root_dir	tool root directory
-v, –verbose	verbose	verbosity level greater than or equal to 0

3.5.2 Per-user configuration file (.dejagnurc)

The per-user configuration file is named `.dejagnurc` in the user's home directory. It is used to customize the behaviour of `runtest` for each user – typically the user's preference for log verbosity, and for storing any experimental Tcl procedures. An example `~/.dejagnurc` file looks like:

Example .dejagnurc

```
set all_flag 1
set RLOGIN /usr/ucb/rlogin
set RSH /usr/local/sbin/ssh
```

Here `all_flag` is set so that I see all the test cases that PASS along with the ones that FAIL. I also set `RLOGIN` to the BSD (non-Kerberos) version. I also set `RSH` to the SSH secure shell, as rsh is mostly used to test Unix machines within a local network.

4 Extending DejaGnu

4.1 Adding a new testsuite

The testsuite for a new tool should always be located in that tools source directory. DejaGnu require the directory be named `testsuite`. Under this directory, the test cases go in a subdirectory whose name begins with the tool name. For example, for a tool named *gdb*, each subdirectory containing testsuites must start with '`gdb.`'.

4.2 Adding 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 testsuites. Unfortunately, well-established testsuites 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 testsuites may make the prospect of creating an entirely new testsuite appear overwhelming. Nevertheless, it is straightforward to start a new testsuite.

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

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

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.ac` for use with GNU Autoconf.

4.2.1 Sample Makefile.in Fragment

Create `Makefile.in` (if using Autoconf), or `Makefile.am` (if using Automake), the source file used by configure to build your `Makefile`. If you are using GNU Automake.just add the keyword *dejagnu* to the *AUTOMAKE_OPTIONS* variable in your `Makefile.am` file. This will add all the `Makefile` support needed to run DejaGnu, and support the Section 2.1 [Make Check], page 6, target.

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. This is called the Section 3.2 [Local config file], page 16, The *check* target must invoke the `runtest` program to run the tests.

The *site.exp* target should usually set up (among other things) the *$tool* variable for the name of your program. If the local `site.exp` file is setup correctly, it is possible to execute the tests by merely typing `runtest` on the command line.

```
# Look for a local version of DejaGnu, otherwise use one in the path
RUNTEST = `if test -f $(top_srcdir)/../dejagnu/runtest; then \
```

```
      echo $(top_srcdir) ../dejagnu/runtest; \
    else \
      echo runtest; \
    fi`

# Flags to pass to runtest
RUNTESTFLAGS =

# Execute the tests
check: site.exp all
        $(RUNTEST) $(RUNTESTFLAGS) --tool ${example} --srcdir $(srcdir)

# Make the local config file
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 ##" >> .
        @cat ./tmp0 > site.exp
        @sed < site.bak \
            -e '1,/^## All variables above are.*##/ d' \
            >> site.exp
        -@rm -f ./tmp?
```

4.2.2 Simple tool init file for batch programs

Create a directory (under `testsuite`) called `config`. Make a tool init file in this directory. Its name must start with the `target_abbrev` value, or be named `default.exp` so call it

`config/unix.exp` for our Unix based example. This is the file that contains the target-dependent procedures. Fortunately, on a native Unix system, most of them do not have to do very much in order for `runtest` to run. If the program being tested is not interactive, you can get away with this minimal `unix.exp` to begin with:

```
proc myprog_exit {} {}
proc myprog_version {} {}
```

If the program being tested is interactive, however, you might as well define a *start* routine and invoke it by using a tool init file like this:

4.2.3 Simple tool init file for interactive programs

```
proc myprog_exit {} {}
proc myprog_version {} {}

proc myprog_start {} {
     global ${examplename}
     spawn ${examplename}
     expect {
-re "" {}
     }
}

# Start the program running we want to test
myprog_start
```

Create a directory whose name begins with your tool's name, to contain tests. For example, if your tool's name is *example*, then the directories all need to start with '`example.`'. Create a sample test file ending in `.exp`. You can use `first-try.exp`. To begin with, just write one line of Tcl code to issue a message:

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

4.2.4 Testing A New Tool Config

Back in the `testsuite` (top level) directory, run `configure`. Typically you do this while in the build directory. You are now ready to type `make check` or `runtest`. You should see something like this:

```
Test Run By bje on Sat Nov 14 15:08:54 AEDT 2015

          === 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 report a test outcome.

Write 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.

4.3 Adding a new target

DejaGnu has some additional requirements for target support, beyond the general-purpose provisions of a `configure` script. DejaGnu 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.

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 using the `verbose` procedure. In cross-development environments, most of the work is in getting the communications right. Code for communicating via TCP/IP networks or serial lines is available in a DejaGnu library files such as `lib/telnet.exp`.

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.

4.4 Adding a new board

Adding a new board consists of creating a new board configuration file. Examples are in `dejagnu/baseboards`. Usually to make a new board file, it's easiest to copy an existing one. It is also possible to have your file be based on a *baseboard* file with only one or two changes needed. Typically, this can be as simple as just changing the linker script. Once the new baseboard file is done, add it to the `boards_DATA` list in the `dejagnu/baseboards/Makefile.am`, and regenerate the Makefile.in using automake. Then just rebuild and install DejaGnu. You can test it by:

There is a crude inheritance scheme going on with board files, so you can include one board file into another, The two main procedures used to do this are `load_generic_config` and `load_base_board_description`. The generic config file contains other procedures used for a certain class of target. The board description file is where the board specific settings go. Commonly there are similar target environments with just different processors.

Testing a New Board Configuration File

```
make check RUNTESTFLAGS="--target_board=newboardfile".
```

Here's an example of a board config file. There are several *helper procedures* used in this example. A helper procedure is one that look for a tool of files in commonly installed locations. These are mostly used when testing in the build tree, because the executables to be tested are in the same tree as the new dejagnu files. The helper procedures are the ones in square braces *[]*, which is the Tcl execution characters.

Example Board Configuration File

```
# Load the generic configuration for this board. This will define a basic
# set of routines needed by the tool to communicate with the board.
load_generic_config "sim"

# basic-sim.exp is a basic description for the standard Cygnus simulator.
load_base_board_description "basic-sim"

# The compiler used to build for this board. This has *nothing* to do
# with what compiler is tested if we're testing gcc.
set_board_info compiler "[find_gcc]"

# We only support newlib on this target.
# However, we include libgloss so we can find the linker scripts.
set_board_info cflags "[newlib_include_flags] [libgloss_include_flags]"
set_board_info ldflags "[newlib_link_flags]"

# No linker script for this board.
set_board_info ldscript "-Tsim.ld"

# The simulator doesn't return exit statuses and we need to indicate this.
set_board_info needs_status_wrapper 1

# Can't pass arguments to this target.
set_board_info noargs 1

# No signals.
set_board_info gdb,nosignals 1

# And it can't call functions.
set_board_info gdb,cannot_call_functions 1
```

4.5 Board configuration file values

The following fields are in the `board_info` array. These are set by the `set_board_info` procedure (or `add_board_info` procedure for appending to lists). Both procedures take a field name and a value for the field (or is added to the field), respectively. Some common board info fields are shown below.

Field	**Example value**	**Description**
compiler	"[find_gcc]"	The path to the compiler to use.
cflags	"-mca"	Compilation flags for the compiler.
ldflags	"[libgloss_link_flags] [newlib_link_flags]"	Linking flags for the compiler.
ldscript	"-Wl,-Tidt.ld"	The linker script to use when cross compiling.
libs	"-lgcc"	Any additional libraries to link in.
shell_prompt	"cygmon>"	The command prompt of the remote shell.
hex_startaddr	"0xa0020000"	The Starting address as a string.
start_addr	0xa0008000	The starting address as a value.

startaddr	"a0020000"	
exit_statuses_bad	1	Whether there is an accurate exit status.
reboot_delay	10	The delay between power off and power on.
unreliable	1	Whether communication with the board is unreliable.
sim	[find_sim]	The path to the simulator to use.
objcopy	$tempfil	The path to the `objcopy` program.
support_libs	"${prefix_dir}/i386-coff/"	Support libraries needed for cross compiling.
addl_link_flags	"-N"	Additional link flags, rarely used.
remotedir	"/tmp/runtest.[pid]"	Directory on the remote target in which executables are downloaded and executed.

These fields are used by the GCC and GDB tests, and are mostly only useful to somewhat trying to debug a new board file for one of these tools. Many of these are used only by a few testcases, and their purpose is esoteric. These are listed with sample values as a guide to better guessing if you need to change any of these.

Board Info Fields For GCC & GDB

Field	Sample Value	Description
strip	$tempfile	Strip the executable of symbols.
gdb_load_offset	"0x40050000"	
gdb_protocol	"remote"	The GDB debugging protocol to use.
gdb_sect_offset	"0x41000000";	
gdb_stub_ldscript	"-Wl,-Teva-stub.ld"	The linker script to use with a GDB stub.
gdb,noargs	1	Whether the target can take command line arguments.
gdb,nosignals	1	Whether there are signals on the target.
gdb,short_int	1	
gdb,target_sim_options	"-sparclite"	Special options to pass to the simulator.
gdb,timeout	540	Timeout value to use for remote communication.
gdb_init_command	"set mipsfpu none"	A single command to send to GDB before the program being debugged is started.
gdb_init_commands	"print/x \$fsr = 0x0"	Same as *gdb_init_command*, except that this is a list, more commands can be added.
gdb_load_offset	"0x12020000"	
gdb_opts	"–command gdbinit"	
gdb_prompt	"\\(gdb960\\)"	The prompt GDB is using.

gdb_run_command	"jump start"	
gdb_stub_offset	"0x12010000"	
use_gdb_stub	1	Whether to use a GDB stub.
wrap_m68k_aout	1	
gcc,no_label_values	1	
gcc,no_trampolines	1	
gcc,no_varargs	1	
gcc,stack_size	16384	Stack size to use with some GCC testcases.
ieee_multilib_flags	"-mieee"	
is_simulator	1	
needs_status_wrapper	1	
no_double	1	
no_long_long	1	
noargs	1	
target_install	{sh-hms}	

4.6 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-oriented and 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 constitutes a test failure. All the C code needed is kept in the test directory. The test driver, written in Tcl, 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` of the GCC testsuite. 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:

- 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.
- Writing (and debugging) the generic Tcl procedures for compilation.
- 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 Tcl procedures specialized for GDB testing.

4.6.1 Hints on writing a test case

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.

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`.

4.7 Debugging a test case

These are the kinds of debugging information available from DejaGnu:

- 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 DejaGnu log file. To do the same for new tests, use the `verbose` procedure (which in turn uses the Tcl variable '`verbose`') to control how much output to generate. This will make it easier for other people running the test to debug it if necessary. If '`verbose`' is zero, there should be no output other than the output from the framework (eg. FAIL). 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 plenty of debugging information.

- 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 Expect. It logs how Expect attempts to match the tool output with the patterns specified. This 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 from Tcl. This logs how Tcl procedure definitions are expanded as they execute. The trace level argument controls the depth of definitions expanded.

- If the value of '`verbose`' is 3 or greater (`runtest -v -v -v`), DejaGnu activates the Expect command `log_user`. This command prints all Expect actions to standard output, to the `.log` file and, if `--debug` is given, to `dbg.log`.

4.8 Adding a test case to a testsuite

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 testsuite subdirectories for the tool of interest to see which approach is most suitable.

Adding a GCC test may be very simple: just add the source file to any test directory beginning with `gcc.` and it will be tested on the next test run.

Adding a test by creating a new directory involves:

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 testsuite subdirectories with the same tool name prefix.
2. Add the new test case to the directory, as above.

4.9 Test case special variables

There are special variables that contain other information from DejaGnu. Your test cases can inspect these variables, as well as the variables saved in `site.exp`. These variables should never be changed.

`$prms_id` The bug tracking system (eg. PRMS/GNATS) number identifying a corresponding bug report (*0* if you do not specify it).

`$bug_id` An optional bug ID, perhaps a bug identification number from another organization (*0* if you do not specify it).

`$subdir` The subdirectory for the current test case.

`$exec_output`
: This is the output from a `${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.

`$expect_out(buffer)`
: The output from the last command. This is an internal variable set by Expect. More information can be found in the Expect manual.

5 Unit testing

5.1 What is unit testing?

Most regression testing as done by DejaGnu is system testing: the complete application is tested all at once. Unit testing is for testing single files, or small libraries. In this case, each file is linked with a test case in C or C++, and each function or class and method is tested in turn, with the test case having to check private data or global variables to see if the function or method worked.

This works particularly well for testing APIs and at level where it is easier to debug them, than by needing to trace through the entire application. Also if there is a specification for the API to be tested, the testcase can also function as a compliance test.

5.2 The dejagnu.h header file

DejaGnu uses a single header file, `dejagnu.h` to assist in unit testing. As this file also produces its one test state output, it can be run stand-alone, which is very useful for testing on embedded systems. This header file has a C and C++ API for the test states, with simple totals, and standardized output. Because the output has been standardized, DejaGnu can be made to work with this test case, without writing almost any Tcl. The library module, dejagnu.exp, will look for the output messages, and then merge them into DejaGnu's.

5.3 C unit testing API

All of the functions that take a `msg` parameter use a C `char *` that is the message to be displayed. There currently is no support for variable length arguments.

- `pass` prints a message for a successful test completion.

  pass(*msg*);
- `fail` prints a message for an unsuccessful test completion.

  fail(*msg*);
- `untested` prints a message for an test case that isn't run for some technical reason.

  untested(*msg*);
- `unresolved` prints a message for an test case that is run, but there is no clear result. These output states require a human to look over the results to determine what happened.

  unresolved(*msg*);
- `totals` prints out the total numbers of all the test state outputs.

  totals();

5.4 C++ unit testing API

All of the methods that take a `msg` parameter use a C char * or STL string, that is the message to be displayed. There currently is no support for variable length arguments.

- `pass` prints a message for a successful test completion.

  TestState::pass(*msg*);

`fail` prints a message for an unsuccessful test completion.

TestState::fail(*msg*);

`untested` prints a message for an test case that isn't run for some reason.

TestState::untested(*msg*);

- `unresolved` prints a message for an test case that is run, but there is no clear result. These output states require a human to look over the results to determine what happened.

  TestState::unresolved(*msg*);

- `totals` prints out the total numbers of all the test state outputs.

  TestState::totals();

6 Reference

6.1 Builtin Procedures

DejaGnu provides these Tcl procedures.

6.1.1 Core Internal Procedures

6.1.1.1 open_logs Procedure

Open the output logs.

open_logs

6.1.1.2 close_logs Procedure

Close the output logs.

close_logs

6.1.1.3 isbuild Procedure

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.

isbuild{*pattern*}

`pattern`

6.1.1.4 is_remote Procedure

Is *board* remote? Return a non-zero value, if so.

is_remote `{` *board* `}`

6.1.1.5 is3way Procedure

Tests for a Canadian cross. This is when the tests will be run on a remotely hosted cross-compiler. If it is a Canadian cross, then the result is *1*; otherwise *0*.

is3way

6.1.1.6 ishost Procedure

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).

ishost{*pattern*}

6.1.1.7 istarget Procedure

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.

istarget `{ args }`

6.1.1.8 isnative Procedure

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*.

isnative

6.1.1.9 log_and_exit Procedure

log_and_exit

6.1.1.10 log_summary Procedure

log_summary`{args}`

`args`

6.1.1.11 setup_xfail Procedure

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.

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

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.

setup_xfail`{config bugid}`

`config` The config triplet to trigger whether this is an unexpected or expect failure.

`bugid` The optional bugid, used to tie this test case to a bug tracking system.

6.1.1.12 pass Procedure

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`.

pass { *message* }

`message` The message to use in the PASS result.

6.1.1.13 fail Procedure

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`.

fail { *message* }

`string` The message to use in the FAIL result.

6.1.1.14 xpass Procedure

Declares a test to have passed when it was expected to fail. `xpass` writes in the log files a message beginning with *XPASS* (or *XFAIL*, if failure was expected), appending the argument `string`.

xpass { *message* }

`message` The message to use in the XPASS result.

6.1.1.15 xfail Procedure

Declares a test to have expectedly failed. `xfail` writes in the log files a message beginning with *XFAIL* (or *PASS*, if success was expected), appending the `message` argument.

xpass { *message* }

6.1.1.16 set_warning_threshold Procedure

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*.

set_warning_threshold{*threshold*}

`threshold`
This is the value of the new warning threshold.

6.1.1.17 get_warning_threshold Procedure

Returns the current value of {`warning_threshold`. The default value is 3. This value controls how many `warning` procedures can be called before becoming *UNRESOLVED*.

get_warning_threshold

6.1.1.18 warning Procedure

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 `perror` 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.

warning { *messsage number* }

`message` The warning message.

`number` The optional number to set the error counter. This is only used to fake out the counter when using the `xfail` procedure to control when it flips the output over to *UNRESOLVED* state.

6.1.1.19 perror Procedure

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

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

perror { *message number* }

`message` The message to be logged.

`number` The optional number to set the error counter. This is only used to fake out the counter when using the `xfail` procedure to control when it flips the output over to *UNRESOLVED* state.

6.1.1.20 note Procedure

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. The `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`.

note { *string* }

`string` The string to use for this note.

6.1.1.21 untested Procedure

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.

untested { *message* }

`message` The message to use.

6.1.1.22 unresolved Procedure

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).

unresolved `{` *`message`* `}`

`string` The message to use.

6.1.1.23 unsupported Procedure

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.

unsupported `{` *`message`* `}`

`message` The message to use.

6.1.1.24 transform Procedure

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: For example, for a cross compiler supporting the *m68k-vxworks* configuration, the result of transform `gcc` is `m68k-vxworks-gcc`.

transform{*`toolname`*}

`toolname` The name of the cross-development program to transform.

6.1.1.25 check_conditional_xfail Procedure

This procedure adds a conditional xfail, based on compiler options used to create a test case executable. If an include options is found in the compiler flags, and it's the right architecture, it'll trigger an *XFAIL*. Otherwise it'll produce an ordinary *FAIL*. You can also specify flags to exclude. This makes a result be a *FAIL*, even if the included options are found. To set the conditional, set the variable `compiler_conditional_xfail_data` to the fields

`"[message string] [targets list] [includes list] [excludes list]"`

(descriptions below). This is the checked at pass/fail decision time, so there is no need to call the procedure yourself, unless you wish to know if it gets triggered. After a pass/fail, the variable is reset, so it doesn't effect other tests. It returns *1* if the conditional is true, or *0* if the conditional is false.

check_conditional_xfail{*`message targets includes excludes`*}

`message` This is the message to print with the normal test result.

`targets` This is a string with the list targets to activate this conditional on.

`includes` This is a list of sets of options to search for in the compiler options to activate this conditional. If the list of sets of options is empty or if any set of the options matches, then this conditional is true. (It may be useful to specify an empty list of include sets if the conditional is always true unless one of the exclude sets matches.)

`excludes` This is a list of sets of options to search for in the compiler options to activate this conditional. If any set of the options matches, (regardless of whether any of the include sets match) then this conditional is de-activated.

Specifying the conditional xfail data

```
set compiler_conditional_xfail_data { \
     "I sure wish I knew why this was hosed" \
            "sparc*-sun*-* *-pc-*-*" \
            {"-Wall -v" "-O3"} \
            {"-O1" "-Map"} \
        }
```

What this does is it matches only for these two targets if "-Wall -v" or "-O3" is set, but neither "-O1" or "-Map" is set. For a set to match, the options specified are searched for independently of each other, so a "-Wall -v" matches either "-Wall -v" or "-v -Wall". A space separates the options in the string. Glob-style regular expressions are also permitted.

6.1.1.26 clear_xfail Procedure

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`.

clear_xfail{*config*}

`config` The system triplets to clear.

6.1.1.27 verbose Procedure

Test cases can use this function to issue helpful messages depending on the number of `-v`/`--verbose` options passed to runtest on the command line. It prints *string* if the value of the variable `verbose` is higher than or equal to the optional *loglevel*. The default log level 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 `-x` argument to log the test results into a parsable XML file. Use the optional `-n` argument to print string without a trailing newline. Use the optional `--` argument if string begins with "-".

verbose{*-log -x -n -r string loglevel*}

`-x`

`-log`

`-n`

`--`

`string`

`number`

6.1.1.28 load_lib Procedure

Loads a DejaGnu library file by searching the default fixed paths built into DejaGnu. 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.

load_lib{*filespec*}

`filespec` The name of the DejaGnu library file to load.

The global variable `libdirs`, handled as a list, is appended to the default fixed paths built into DejaGnu.

Additional search directories for load_lib

```
# append a non-standard search path
    global libdirs
    lappend libdirs $srcdir/../../gcc/testsuite/lib
    # now loading $srcdir/../../gcc/testsuite/lib/foo.exp works
    load_lib foo.exp
```

6.1.2 Procedures For Remote Communication

`lib/remote.exp` defines procedures for establishing and managing communications. 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 ssh, tip, kermit, telnet, rsh, and rlogin. If the `--reboot` option was used on the runtest command line, then the target is rebooted before the connection is made.

6.1.2.1 call_remote Procedure

call_remote{*type proc dest args*}

`proc`

`dest`

`args`

6.1.2.2 check_for_board_status Procedure

check_for_board_status{*variable*}

`variable`

6.1.2.3 file_on_build Procedure

file_on_build{*op file args*}

`op`

`file`

`args`

6.1.2.4 file_on_host Procedure

file_on_host{*op file args*}

op

file

args

6.1.2.5 local_exec Procedure

local_exec{*commandline inp outp timeout*}

inp

outp

timeout

6.1.2.6 remote_binary Procedure

remote_binary{*host*}

host

6.1.2.7 remote_close Procedure

remote_close{*shellid*}

shellid This is the 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.

6.1.2.8 remote_download Procedure

remote_download{*dest file args*}

dest

file

args

6.1.2.9 remote_exec Procedure

remote_exec{*hostname program args*}

hostname

program

args

6.1.2.10 remote_expect Procedure

remote_expect{*board timeout args*}

board

timeout

args

6.1.2.11 remote_file Procedure

remote_file{*dest args*}

dest

args

6.1.2.12 remote_ld Procedure

remote_ld{*dest prog*}

dest

prog

6.1.2.13 remote_load Procedure

remote_load{*dest prog args*}

dest

prog

args

6.1.2.14 remote_open Procedure

remote_open{*type*}

type
: 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.

6.1.2.15 remote_pop_conn Procedure

remote_pop_conn{*host*}

host

6.1.2.16 remote_push_conn Procedure

remote_push_conn{*host*}

host

6.1.2.17 remote_raw_binary Procedure

remote_raw_binary{*host*}

host

6.1.2.18 remote_raw_close Procedure

remote_raw_close{*host*}

host

6.1.2.19 remote_raw_file Procedure

remote_raw_file{*dest args*}

dest

args

6.1.2.20 remote_raw_ld Procedure

remote_raw_ld{*dest prog*}

dest

prog

6.1.2.21 remote_raw_load Procedure

remote_raw_load{*dest prog args*}

dest

prog

args

6.1.2.22 remote_raw_open Procedure

remote_raw_open{*args*}

args

6.1.2.23 remote_raw_send Procedure

remote_raw_send{*dest string*}

dest

string

6.1.2.24 remote_raw_spawn Procedure

remote_raw_spawn{*dest commandline*}

dest

commandline

6.1.2.25 remote_raw_transmit Procedure

remote_raw_transmit{*dest file*}

dest

file

6.1.2.26 remote_raw_wait Procedure

remote_raw_wait{*dest timeout*}

dest

timeout

6.1.2.27 remote_reboot Procedure

Return value of this function depends on actual implementation of reboot that will be used, in practice it is expected that `remote_reboot` returns *1* on success and *0* on failure.

remote_reboot{*host*}

`host`

6.1.2.28 remote_send Procedure

remote_send{*dest string*}

`dest`

`string`

6.1.2.29 remote_spawn Procedure

remote_spawn{*dest commandline args*}

`dest`

`commandline`
`args`

6.1.2.30 remote_swap_conn Procedure

remote_swap_conn{*host*}

6.1.2.31 remote_transmit Procedure

remote_transmit{*dest file*}

`dest`

`file`

6.1.2.32 remote_upload Procedure

remote_upload{*dest srcfile arg*}

`dest`

`srcfile`

`arg`

6.1.2.33 remote_wait Procedure

remote_wait{*dest timeout*}

`dest`

`timeout`

6.1.2.34 standard_close Procedure

standard_close{*host*}

`host`

6.1.2.35 standard_download Procedure

standard_download{*dest file destfile*}

dest

file

destfile

6.1.2.36 standard_exec Procedure

standard_exec{*hostname args*}

hostname

args

6.1.2.37 standard_file Procedure

standard_file{*dest op args*}

6.1.2.38 standard_load Procedure

standard_load{*dest prog args*}

dest

prog

args

6.1.2.39 standard_reboot Procedure

It looks like that this procedure is never called, instead `${board}_reboot` defined in `base-config.exp` will be used because it has higher priority and `base-config.exp` is always imported by `runtest`.

standard_reboot{*host*}

host

6.1.2.40 standard_send Procedure

standard_send{*dest string*}

dest

string

6.1.2.41 standard_spawn Procedure

standard_spawn{*dest commandline*}

dest

commandline

6.1.2.42 standard_transmit Procedure

standard_transmit{*dest file*}

`dest`

`file`

6.1.2.43 standard_upload Procedure

standard_upload{*dest srcfile destfile*}

`dest`

`srcfile`

`destfile`

6.1.2.44 standard_wait Procedure

standard_wait{*dest timeout*}

`dest`

`timeout`

6.1.2.45 unix_clean_filename Procedure

unix_clean_filename{*dest file*}

`dest`

`file`

6.1.3 Procedures For Using Utilities to Connect

6.1.3.1 telnet Procedure

telnet{*hostname port*}

rlogin{*hostname*}

6.1.3.2 rsh Procedure

rsh{*hostname*}

`hostname` This 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, separated by a *:*. If host or target is used in the `hostname` field, than the config array is used for all information.

6.1.3.3 tip Procedure

tip{*port*}

`port` 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. the config array is used for all information.

6.1.3.4 kermit Procedure

kermit{*port bps*}

port — Connect using the program `kermit`. `Port` is the device name, e.g. `/dev/ttyb`.

bps — `bps` is the line speed to use (in its 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. the config array is used for all information.

6.1.3.5 kermit_open Procedure

kermit_open{*dest args*}

dest

args

6.1.3.6 kermit_command Procedure

kermit_command{*dest args*}

dest

args

6.1.3.7 kermit_send Procedure

kermit_send{*dest string args*}

dest

string

args

6.1.3.8 kermit_transmit Procedure

kermit_transmit{*dest file args*}

dest

file

args

6.1.3.9 telnet_open Procedure

telnet_open{*hostname args*}

hostname

args

6.1.3.10 telnet_binary Procedure

telnet_binary{*hostname*}

hostname

6.1.3.11 telnet_transmit Procedure

telnet_transmit{*dest file args*}

dest

file

args

6.1.3.12 tip_open Procedure

tip_open{*hostname*}

hostname

6.1.3.13 rlogin_open Procedure

rlogin_open{*arg*}

arg

6.1.3.14 rlogin_spawn Procedure

rlogin_spawn{*dest cmdline*}

dest

cmdline

6.1.3.15 rsh_open Procedure

rsh_open{*hostname*}

hostname

6.1.3.16 rsh_download Procedure

rsh_download{*desthost srcfile destfile*}

desthost

srcfile

destfile

6.1.3.17 rsh_upload Procedure

rsh_upload{*desthost srcfile destfile*}

desthost

srcfile

destfile

6.1.3.18 rsh_exec Procedure

rsh_exec{*boardname cmd args*}

boardname
cmd

args

6.1.3.19 ssh_close procedure

ssh_close {*desthost*}

desthost

6.1.3.20 ssh_exec procedure

ssh_exec{*boardname program pargs inp outp*}

boardname
program

pargs

inp

outp

6.1.3.21 ssh_download procedure

ssh_download{*desthost srcfile destfile*}

desthost

srcfile

destfile

6.1.3.22 ssh_upload procedure

ssh_upload{*desthost srcfile destfile*}

desthost

srcfile

destfile

6.1.3.23 ftp_open Procedure

ftp_open{*host*}

host

6.1.3.24 ftp_upload Procedure

ftp_upload{*host remotefile localfile*}

host

remotefile
localfile

6.1.3.25 ftp_download Procedure

ftp_download{*host localfile remotefile*}

`host`

`localfile`
`remotefile`

6.1.3.26 ftp_close Procedure

ftp_close{*host*}

`host`

6.1.3.27 tip_download Procedure

tip_download{*spawnid file*}

`spawnid` 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.

`file` This is the filename to download.

6.1.4 Procedures For Target Boards

6.1.4.1 default_link Procedure

default_link{*board objects destfile flags*}

`board`

`objects`

`destfile`

`flags`

6.1.4.2 default_target_assemble Procedure

default_target_assemble{*source destfile flags*}

`source`

`destfile`

`flags`

6.1.4.3 default_target_compile Procedure

default_target_compile{*source destfile type options*}

`source`

`destfile`

`type`

`options`

6.1.4.4 pop_config Procedure

pop_config{*type*}

type

6.1.4.5 prune_warnings Procedure

prune_warnings{*text*}

text

6.1.4.6 push_build Procedure

push_build{*name*}

name

6.1.4.7 push_config Procedure

push_config{*type name*}

type

name

6.1.4.8 reboot_target Procedure

Reboot the target.

reboot_target

6.1.4.9 target_assemble Procedure

target_assemble{*source destfile flags*}

source

destfile

flags

6.1.4.10 target_compile Procedure

target_compile{*source destfile type options*}

source

destfile

type

options

6.1.5 Target Database Procedures

6.1.5.1 board_info Procedure

board_info{*machine op args*}

machine

op

args

6.1.5.2 host_info Procedure

host_info{*op args*}

`op`

`args`

6.1.5.3 set_board_info Procedure

This checks if `board_info` array's field *entry* has been set already and if not, then sets it to *value*.

set_board_info{*entry value*}

`entry` The name of a `board_info` field to operate on.

`value` The value to set the field to.

6.1.5.4 add_board_info Procedure

This treats `board_info` array's field *entry* as a TCL list and adds *value* at the end.

add_board_info{*entry value*}

`entry` The name of a `board_info` field to operate on.

`value` The value to add to the field.

6.1.5.5 set_currtarget_info Procedure

set_currtarget_info{*entry value*}

`entry`

`value`

6.1.5.6 target_info Procedure

target_info{*op args*}

`op`

`args`

6.1.5.7 unset_board_info Procedure

This checks if `board_info` array's field *entry* has been set and if so, then removes it.

unset_board_info{*entry*}

`entry` The name of a `board_info` field to operate on.

6.1.5.8 unset_currtarget_info Procedure

unset_currtarget_info{*entry*}

`entry`

6.1.5.9 push_target Procedure

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.

push_target{*name*}

`name` The name of the target to make current connection.

6.1.5.10 pop_target Procedure

This unsets the current target connection.

pop_target

6.1.5.11 list_targets Procedure

This lists all the supported targets for this architecture.

list_targets

6.1.5.12 push_host Procedure

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.

push_host{*name*}

`name`

6.1.5.13 pop_host Procedure

This unsets the current host connection.

pop_host

6.1.5.14 compile Procedure

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`.

compile{*file*}

`file`

6.1.5.15 archive Procedure

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`.

archive{*file*}

`file`

6.1.5.16 ranlib Procedure

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.

ranlib{*file*}

`file`

6.1.5.17 execute_anywhere Procedure

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.

execute_anywhere{*cmdline*}

`cmdline`

6.1.6 Platform 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 underscore _, and finally a suffix describing the procedure's purpose. For example, a procedure to extract the version from GDB is called `gdb_version`.

`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 testsuite.

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.

6.1.6.1 ${tool}_start Procedure

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.

DejaGnu 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}_start

6.1.6.2 ${tool}_load Procedure

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}_load

6.1.6.3 ${tool}_exit Procedure

Cleans up (if necessary) before DejaGnu 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}_exit

6.1.6.4 ${tool}_version Procedure

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.

${tool}_version

6.1.7 Utility Procedures

6.1.7.1 getdirs Procedure

Returns a list of all the directories in the single directory a single directory that match an optional pattern.

getdirs{*rootdir pattern*}

`args`

`pattern` If you do not specify `pattern`, `Getdirs` assumes a default pattern of *. You may use the common shell wildcard characters in the pattern. If no directories match the pattern, then a NULL string is returned.

6.1.7.2 find Procedure

Search for files whose names match *pattern* (using shell wildcard characters for filename expansion). Search subdirectories recursively, starting at *rootdir*. 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.

find{*rootdir pattern*}

`rootdir` The top level directory to search the search from.

`pattern` A csh "glob" style regular expression representing the files to find.

6.1.7.3 which Procedure

Searches the execution path for an executable file *binary*, like 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`.

which{*file*}

`binary` The executable program or shell script to look for.

6.1.7.4 grep Procedure

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`.)

grep{*filename regexp --line*}

`filename` The file to search.

`regexp` The Unix style regular expression (as used by the `grep` Unix utility) to search for.

`--line` Prefix the line number to each line where the regexp matches.

6.1.7.5 prune Procedure

This procedure is deprecated and will be removed in the next release of DejaGnu. If a testsuite uses this procedure, a copy of the procedure should be made and placed in the lib directory of the testsuite.

6.1.7.6 runtest_file_p Procedure

Search *runtests* for *testcase* and return *1* if found, *0* if not. *runtests* is a list of two elements. The first 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. The second is the pathname of the current testcase under consideration. This is used by tools like compilers where each testcase is a file.

runtest_file_p{*runtests testcase*}

`runtests` The list of patterns to compare against.

`testcase` The test case filename.

6.1.7.7 diff Procedure

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.

diff{*file_1 file_2*}

`file_1` The first file to compare.

`file_2` The second file to compare.

6.1.7.8 setenv Procedure

Sets the environment variable *var* to the value *val*.

setenv{*var val*}

`var` The environment variable to set.

`val` The value to set the variable to.

6.1.7.9 unsetenv Procedure

Unsets the environment variable *var*.

unsetenv{*var*}

`var` The environment variable to unset.

6.1.7.10 getenv Procedure

Returns the value of *var* in the environment if it exists, otherwise it returns NULL.

getenv{*var*}

`var` The environment variable to get the value of.

6.1.7.11 prune_system_crud Procedure

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.

prune_system_crud{*system test*}

`system` The system error messages to look for to screen out.

`text` The Tcl variable containing the text.

6.1.8 Libgloss, a free board support package (BSP)

Libgloss is a free *BSP* (Board Support Package) commonly used with GCC and G++ to produce a fully linked executable image for an embedded systems.

6.1.8.1 libgloss_link_flags Procedure

libgloss_link_flags{*args*}

`args`

6.1.8.2 libgloss_include_flags Procedure

libgloss_include_flags{*args*}

`args`

6.1.8.3 newlib_link_flags Procedure

newlib_link_flags{*args*}

`args`

6.1.8.4 newlib_include_flags Procedure

newlib_include_flags{*args*}

args

6.1.8.5 libio_include_flags Procedure

libio_include_flags{*args*}

args

6.1.8.6 libio_link_flags Procedure

libio_link_flags{*args*}

args

6.1.8.7 g++_include_flags Procedure

g++_include_flags{*args*}

args

6.1.8.8 g++_link_flags Procedure

g++_link_flags{*args*}

args

6.1.8.9 libstdc++_include_flags Procedure

libstdc++_include_flags{*args*}

args

6.1.8.10 libstdc++_link_flags Procedure

libstdc++_link_flags{*args*}

args

6.1.8.11 get_multilibs Procedure

get_multilibs{*args*}

args

6.1.8.12 find_binutils_prog Procedure

find_binutils_prog{*name*}

name

6.1.8.13 find_gcc Procedure

find_gcc

6.1.8.14 find_gcj Procedure

find_gcj

6.1.8.15 find_g++ Procedure

find_g++

6.1.8.16 find_g77 Procedure

find_g77

6.1.8.17 find_gfortran Procedure

find_gfortran

6.1.8.18 process_multilib_options Procedure

process_multilib_options{*args*}

`args`

6.1.8.19 add_multilib_option Procedure

add_multilib_option{*args*}

`args`

6.1.8.20 find_gas Procedure

find_gas

6.1.8.21 find_ld Procedure

find_ld

6.1.8.22 build_wrapper Procedure

build_wrapper{*gluefile*}

`gluefile`

6.1.8.23 winsup_include_flags Procedure

winsup_include_flags{*args*}

`args`

6.1.8.24 winsup_link_flags Procedure

winsup_link_flags{*args*}

`args`

6.1.9 Procedures for debugging your scripts

`lib/debugger.exp` defines the following procedures:

6.1.9.1 dumpvars Procedure

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*.

dumpvars{*vars*}

`vars` The variables to dump.

6.1.9.2 dumplocals Procedure

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*.

dumplocals{*args*}

`args`

6.1.9.3 dumprocs Procedure

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

dumprocs{*pattern*}

`pattern` The csh "glob" style pattern to look for.

6.1.9.4 dumpwatch Procedure

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

dumpwatch{*pattern*}

`pattern` The csh "glob" style pattern to look for.

6.1.9.5 watcharray Procedure

watcharray{*element type*}

`type` The csh "glob" style pattern to look for.

6.1.9.6 watchvar Procedure

watchvar{*var type*}

6.1.9.7 watchunset Procedure

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

watchunset{*arg*}

`args`

6.1.9.8 watchwrite Procedure

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

watchwrite{*var*}

`var` The variable to watch.

6.1.9.9 watchread Procedure

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

watchread{*var*}

`var` The variable to watch.

6.1.9.10 watchdel Procedure

This deletes a watchpoint from the watch list. It is abbreviated as *wd*.

watchdel{*args*}

`args`

6.1.9.11 print Procedure

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

print{*var*}

`var`

6.1.9.12 quit Procedure

This makes runtest exit. It is abbreviated as *q*.

quit