DejaGnu

Version 1.6.3

28 December 2020

Rob Savoye et al.
Cygnus Support and the GNU Project

Table of Contents

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.
- A test script aborts due to a Tcl error. In this case, the DejaGnu framework inserts an *UNRESOLVED* result as a placeholder for an unknown number of tests that were not run because the script crashed.

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 4.2 [Local configuration file], page 18, 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. See Section "DejaGnu Tests" in *The GNU Automake Manual*.

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-manufacturer-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 `Ctrl-c`).

`--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`.

`--global_init [name]`
Use *name* as the global init file instead of `site.exp` in *libdir*. The default is, of course, `site.exp`. Note that this option accepts a relative file name, interpreted starting at *libdir*, so a file in a subdirectory may be used. This is probably less useful for most sites, but is orthogonal with the `--local_init` option and may be useful in large testing labs.

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

`--local_init [name]`
Use *name* as the testsuite local init file instead of `site.exp` in the current directory and in *objdir*. The default is, of course, `site.exp`. Note that this option accepts a relative file name, so a file in a subdirectory may be used.

`--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-manufacturer-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`
Generate XML output. The output file is named after the tool with an .xml extension.

`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* scripts that control them. *testsuite*.exp cannot include directory names, only plain filenames.

`arg(s)` specifies a subset of test cases to run. For compiler or assembler tests, which often use a single *.exp* script covering many different test case files, this option allows you to further restrict the tests by listing particular test cases. For larger testsuites such as that included in GCC, this can save a lot of time. Some tools support wildcards here, but this varies from tool to tool. Typically the wildcards *?*, ***, 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 Running other DejaGnu commands

DejaGnu now features auxiliary commands not directly related to running tests, but somehow related to the broader purpose of testing.

These commands are run via the `dejagnu` multiplex launcher, which locates an appropriate script and the required interpreter and then runs the requested command.

3.1 Invoking `dejagnu`

The `dejagnu` launcher is primarily designed to pass most options on to the scripts that it runs, but does process the `--help` and `--version` options entirely internally, while also recognizing the `--verbose` option.

```
dejagnu <command> [options...]
dejagnu --help
dejagnu --version
```

Note that the command names may contain multiple words. In this case, the command can be given as separate arguments to `dejagnu` or combined with dashes ('`-`'); both forms are equivalent.

All words of the command name must appear before any options. The search for a command terminates when an option is found.

Note that the first valid command found is used. A longer command name can be shadowed by a shorter command name that happens to be a prefix of the longer name, if the command name is given as multiple arguments. The equivalent form with the longer command name combined using dashes into a single argument will correctly refer to the otherwise shadowed command.

The `dejagnu` launcher can also be run using symbolic links, provided that the shell places the name under which `dejagnu` was invoked in `$0`. Any dash-separated words after "dejagnu" in the name of such a link are taken to be the leading words of a command name.

The `dejagnu` launcher supports alternate implementations depending upon available interpreters.

Options for the `dejagnu` launcher itself cannot be abbreviated, since the launcher has no way to know which abbreviations are unique and which would be ambiguous to the invoked command.

`--help` Print a help message instead of running a command.

`-V`, `--version`
: Print a version banner for the launcher itself including the version of DejaGnu. Any command given is ignored.

`-v`, `--verbose`
: Emit additional output describing the inner workings of the `dejagnu` launcher. This option is also passed on to the invoked command.

All arguments after the command name are passed to the invoked command.

3.2 Invoking `dejagnu help`

The `dejagnu help` tool displays long-form documentation for DejaGnu auxiliary commands that are invoked using the `dejagnu` launcher.

```
dejagnu help [options...] <command>
```

Again, command names may contain multiple words. This command forms an operand by joining all words in the command name using dashes ('`-`') and prepending '`dejagnu-`'. This is then used as the name of a manual page and passed to the `man` command.

If the manual page is in a particular directory relative to the script implementing this command, a full file name is produced, otherwise, `man` performs its normal search.

The `--verbose` option causes additional output describing the inner workings of the `dejagnu help` command to be produced.

The `--path`, `-w`, and `-W` options are passed to `man`.

3.3 Invoking `dejagnu report card`

The `dejagnu report card` tool produces a tabular summary of the results from test runs by reading the summary files that DejaGnu produces.

```
dejagnu report card [<option>|<tool>|<file>]...
```

The `--verbose` option causes additional output describing the inner workings of the `dejagnu report card` command to be produced.

Aside from options, the command may include a list of tools or files. Names ending in '`.sum`' are used as-is. Names ending in '`.log`' are changed to instead refer to the summary file. Names ending with a simple dot ('`.`') have '`sum`' appended, for convenience when using Readline filename completion in a shell, which will complete to the dot, since there are both '`.sum`' and '`.log`' files produced for each tool tested. Lastly, all other names are taken as tool names and '`.sum`' is appended to refer to the summary file produced by DejaGnu.

The relevant summary files are read and an ASCII-art table is produced. The table has columns for counts of tests passed, failed, unsupported, unresolved, and untested. Tests that are expected to pass and tests that are expected to fail are counted in separate columns, but known failures ('`KFAIL`' and '`KPASS`') are summarized together with expected failures ('`XFAIL`' and '`XPASS`') in two additional columns: '`?PASS`' and '`?FAIL`'. Additionally, if a test produced any warnings or errors, tags '`!W!`' or '`!E!`' are added at the end of the row.

4 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. If `DEJAGNU` is set, but the file cannot be located, an error will be raised and `runtest` will abort.

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.

4.1 Global configuration 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). This example is based on a configuration once used at Cygnus.

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

switch -glob -- $target_triplet {
    "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 5.4 [Adding a new board], page 28, 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.

4.2 Local configuration 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 configuration 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 Configuration 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 $srcdir/../libg++/src -isystem $srcdir/../libio -isystem $srcdir/../libstdc++ -isystem $srcdir/../libstdc++/stl -L/build/devo-builds/i686-pc-linux-gnu/gcc/../libg++ -L/build/devo-builds/i686-pc-linux-gnu/gcc/../libstdc++"
append LDFLAGS " -L/build/devo-builds/i686-pc-linux-gnu/gcc/../ld"
set tmpdir /build/devo-builds/i686-pc-linux-gnu/gcc/testsuite
set srcdir "${srcdir}/testsuite"
## All variables above are generated by configure. Do Not Edit ##
```

This file defines the required fields for a local configuration 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.

4.3 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, see Section 5.4 [Adding a new board], page 28.

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. DejaGnu will use this procedure to locate a compiler if the `compiler` field is not set.
- `libgloss_include_flags` returns the flags to compile using Section A.8 [Libgloss], page 66, the GNU board support package (BSP).
- `libgloss_link_flags` returns the flags to link an executable using Section A.8 [Libgloss], page 66.
- `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).

4.4 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.

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" > check.out 2>&1
```

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=921202-1.c"
```

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.

4.5 Configuration 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.

4.5.1 Command line option variables

In the user editable second section of the Section 4.5.2 [User configuration file], page 23, you can not only override the configuration variables captured in the first section, but also specify default values for all of the `runtest` command line options. Excepting `--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`. Refer to 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
–global_init	global_init_file	file name for global init file in `libdir`
–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
–local_init	local_init_file	file name for local init file in `objdir`
–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 (selects tests to run)
–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

4.5.2 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.

5 Extending DejaGnu

This chapter describes how to extend DejaGnu with new testsuites, new tools, new targets and new boards.

5.1 Adding a new testsuite

The testsuite for a new package should always be located in the source directory of that package. DejaGnu requires this directory to be named `testsuite`. Under this directory, the test cases go in various subdirectories whose name begins with the tool name. The organization of the various testsuite subdirectories is up to you. For a tool named `gdb`, for instance, each subdirectory containing tests must start with '`gdb.`'.

5.2 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.

5.2.1 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 4.2 [Local configuration file], page 18, 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 ##" >> ./tmp0
        @cat ./tmp0 > site.exp
        @sed < site.bak \
            -e '1,/^## All variables above are.*##/ d' \
            >> site.exp
        -@rm -f ./tmp?
```

5.2.2 Simple tool init file for batch programs

The tool init file may be placed in `testsuite/lib` or in `testsuite/lib/tool` and must be named `*tool*.exp`, where *tool* is the name of the tool to be tested. For this example, we will use the name '`example`' for the tool name, which means that the tool init file must be named `example.exp`. If the program being tested is not interactive, you can get away with this minimal tool init file to begin with:

```
proc example_exit {} {}
proc example_version {} {}
```

By convention, the file name for the executable for a tool should be stored in a global variable with the same name as the tool, but in all uppercase. For our example program '`example`', the name of the program under test should be stored in '`EXAMPLE`'.

5.2.3 Simple tool init file for interactive programs

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:

```
proc example_exit {} {}
proc example_version {} {}

proc example_start {} {
     global EXAMPLE
     spawn $EXAMPLE
     expect {
        -re "" {}
     }
}

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

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

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

5.2.4 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.

5.3 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.

5.4 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 configuration 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 configuration 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 brackets, which indicates a Tcl procedure call.

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. Further, this is
# the default, so this line is optional for most boards.
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
```

5.5 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}	

5.6 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.

5.6.1 Hints on writing a test case

To preserve basic sanity, no should 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 `perror` or `warning`. Note that using `perror` will cause the next text result to be reported as '`UNRESOLVED`', so printing an error and allowing the test to fail is a good option.

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.

If you use glob patterns, you will need to escape any '`*`', '`?`', '`[`', '`]`', and '`\`' characters that are meant to match literally. If you use regular expressions, see the *re_syntax(n)* manual page from Tcl for the syntax details, and be sure to understand what punctuation characters match literally and what characters have special meanings in regular expressions.

Tcl has a few options for quoting; the most notable are '`{}`' and '`""`'. These quotes behave differently: '`{}`' must balance, while '`""`' performs various interpolations. In '`{}`' quotes, unbalanced '`{`' or '`}`' characters must be escaped with '`\`' and these escapes are *not* removed; fortunately, backslash-escaped braces match literal braces in Tcl regular expressions. In '`""`' quotes, any embedded '`"`' characters must be escaped, a literal '`$`' begins a variable substitution, and unescaped '`[]`' introduce a Tcl command substitution.

Synchronization with the tested program

A DejaGnu testsuite executes concurrently with the programs that it tests. As a result, DejaGnu may see parts of the tested program's output while the tested program is still producing more output. Expect patterns must be written to handle the possibility that incomplete output from the tested program will be considered for matching.

Expect reads the output from the tested program into an internal matching buffer and removes everything from the start of the buffer to the end of the match when a match is found. Any given character can be matched at most once, or skipped if a match is found starting later in the buffer or the buffer reaches its capacity. Anything left in the buffer after the end of the match remains in the buffer and is considered for the next `expect` command. If `expect` is invoked and no patterns match, Expect waits for more text to arrive. New text is appended to the buffer as it is read. If the buffer reaches its capacity, the entire contents of the buffer are discarded and Expect resumes reading.

In Expect patterns, the regular expression anchors '^' and '$' match at the beginning and end of the *buffer*, not at line boundaries. The '$' anchor must be used with care because it will match at the end of what Expect *has* read, but the program may have produced more output that Expect *has not yet* read. Similarly, regular expressions ending with the '*' quantifier can potentially match a prefix of the intended text, only for the rest to arrive shortly thereafter.

Maintaining synchronization with the tested program is easier if the patterns match all of the output generated by the tested program; this is called closure.

For interactive programs, a prompt is usually a good synchronization point, provided that the program's prompt can be uniquely recognized. Since the prompt is usually the last output until the program receives further input, the '$' anchor can be useful here.

If the output from the tested program is organized into lines, matching end-of-line using '\n' is usually a good way to process one line at a time. Note that terminal settings may result in the insertion of additional '\r' characters, usually translating '\n' to '\r\n'.

Be careful not to neglect output generated by setup rather than by the interesting parts of a test case. For example, while testing GDB, a 'set height 0\n' command is issued. 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 '(gdb) ' prompt will remain in the buffer and begin the text seen by the next `expect` command—which might make that pattern fail to match.

Priority of Expect patterns

Be particularly careful about how you write the patterns. Expect attempts to match each pattern in the order that they are written in the `expect` command. Unless a regexp pattern is anchored at the beginning of the buffer, Expect can search ahead for a match for a pattern that appears earlier in the `expect` command and skip over text that would match a later pattern. *The text thus skipped is discarded.* This is a source of very hard to trace bugs, especially when reading input from batch-oriented unit tests.

For example, consider a simple model once used by the DejaGnu testsuite for unit testing. In this example, a test has failed, but the tests before and after it have passed. First the relevant input to DejaGnu:

```
PASSED: foo
FAILED: bar
PASSED: baz
```

The test script is reading this with two Expect patterns, simplified for this example by omitting handling of the actual messages and other possible results:

```
expect {
        -re {PASSED: [^\r\n]+[\r\n]+} { pass ... }
        -re {FAILED: [^\r\n]+[\r\n]+} { fail ... }
}
```

At every cycle, Expect attempts to match each pattern in the order that they are written against the available input. If DejaGnu is processing the input as quickly as it arrives, this example will actually work. However, if the system scheduler sets DejaGnu aside for a bit, or the external program produces output in a burst, Expect can find that its input buffer contains the text in the first example above all at once as the cycle begins.

If this occurs, Expect will first attempt to match `{PASSED: [^\r\n]+[\r\n]+}` against the input and will succeed, since the input begins with '`PASSED: foo`'. The `pass` procedure is called and the test result recorded. Expect then starts a new matching cycle.

If the input had been presented one line at a time, the expected result would occur: the `{FAILED: [^\r\n]+[\r\n]+}` pattern would match and the test driver would work correctly. But we are considering the case where all three lines arrived "at once" so we must examine what Expect will do in this case. After the first line has been processed, the Expect buffer now contains:

```
FAILED: bar
PASSED: baz
```

Expect again attempts to match each pattern in order. Expect will attempt to match `{PASSED: [^\r\n]+[\r\n]+}` before attempting to match `{FAILED: [^\r\n]+[\r\n]+}` and the first attempt succeeds because the pattern is not anchored. The '`FAILED: bar`' message is simply discarded when Expect finds the later '`PASSED:baz`' message in the buffer.

How to prevent this? There are two ways: either group all of your test matches into a single regexp using alternation, or ensure that all patterns can match only at the start of Expect's buffer. Both options can be made to work. Grouping all status results into a single regexp allows some other unspecified text to still be silently discarded, while ensuring that all patterns are anchored absolutely requires closure, as any unmatched text will cause Expect to run out of buffer space. Expect discards the entire buffer when this occurs.

5.7 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`.

5.8 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.

5.9 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.

6 Unit testing

6.1 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 at a 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.

6.2 Running unit tests

The native DejaGnu unit testing support is provided by a library module `dejagnu.exp` and the procedure `host_execute` is called by testsuite code to run unit tests.

host_execute *program arguments*

The `host_execute` procedure runs *program*, passing *arguments* on the command line, and examines the output for test result messages according to the DejaGnu unit testing protocol.

If successful, the return value is an empty string. Otherwise, an error message is returned.

6.3 DejaGnu unit test protocol

DejaGnu spawns a unit test program and reads that program's output. Arguments for the unit test program can be specified in the testsuite.

Unit test programs may produce any output for the benefit of a developer running them directly or reading the DejaGnu log, but output matching the Tcl regexp `{\n\t[][[:upper:]]*:}` (a tab character at the beginning of a line, followed by any sequence of uppercase letters and square brackets, followed by a colon) should be considered reserved for future extension. Future versions of DejaGnu will interpret more lines matching this pattern as additional unit test information.

⊣ `NOTE:` *text*

This will cause *text* to be printed at verbose levels 2 and higher.

⊣ `PASSED:` *name*

⊣ `FAILED:` *name*

⊣ `XPASSED:` *name*

⊣ `XFAILED:` *name*

⊣ `UNTESTED:` *name*

⊣ `UNRESOLVED:` *name*

These indicate simple test results.

Note that all output from a unit test program, including the lines recognized and interpreted by DejaGnu, appears in the log.

6.4 C unit testing API

The C API is provided in the `dejagnu.h` header file. This header provides a self-contained implementation. For convenience, the `totals()` function can be called at the end of the unit test program to output summary totals. DejaGnu counts the test results independently and will operate correctly even if `totals()` is never invoked.

All of the functions that take a `msg` parameter use a C `char *` that is the message to be displayed. All of the functions that display a message accept a `printf`-style format string and variable arguments.

- `note` emits a note that will be displayed at verbose level 2 or higher.

  note(*msg*, ...);
- `pass` prints a message for a successful test completion.

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

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

  xfail(*msg*, ...);
- `xpass` prints a message for an unexpected successful test completion.

  xpass(*msg*, ...);
- `untested` prints a placeholder message for a test case that is not yet implemented or that could not be run for some reason.

  untested(*msg*, ...);
- `unresolved` prints a message for a test case that was run, but did not produce a clear result. These output states require a human to look over the results to determine what happened.

  unresolved(*msg*, ...);
- `totals` prints out the total counts of all of the test results as a convenience when running the unit test program directly. DejaGnu does not use this information and instead counts the results independently.

  totals();

6.5 C++ unit testing API

The C++ API is also provided in the `dejagnu.h` header file. This header provides a self-contained implementation. For convenience, the `totals()` method outputs summary totals to be used at the end of unit test program. DejaGnu does not depend on this summary and counts the test results independently.

All of the methods that take a `msg` parameter use a STL string as the message to be displayed. There currently is no support for formatted output in the C++ API; build the desired string before passing it to these functions.

Note that the C API is also available in C++ unit test programs; using both will cause confusion because each `TestState` object carries its own set of summary counters, while the C API has an independent global set of summary counters.

The `TestState` class supports the following instance methods:

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

  TestState::pass (*msg*) ;

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

  TestState::fail (*msg*) ;

- `xfail` prints a message for an expected unsuccessful test completion.

  TestState::xfail (*msg*) ;

- `xpass` prints a message for an unexpected successful test completion.

  TestState::xpass (*msg*) ;

- `untested` prints a placeholder message for a test case that is not yet implemented or that could not be run for some reason.

  TestState::untested (*msg*) ;

- `unresolved` prints a message for a test case that was run, but did not produce a 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 counts of all of the test results as a convenience when running the unit test program directly. DejaGnu does not use this information and instead counts the results independently.

  In the C++ API, this method is automatically called when a `TestState` instance is destroyed.

  TestState::totals () ;

Appendix A Built-in Procedures

DejaGnu provides these Tcl procedures.

A.1 Core Internal Procedures

open_logs Procedure

Open the output logs.

open_logs

close_logs Procedure

Close the output logs.

close_logs

isbuild Procedure

Tests for a particular build host environment. If the currently configured host matches the `pattern` argument, the result is *1*; otherwise the result is *0*. *pattern* must be a full three-part configure triplet; in particular, you may not use the shorter aliases supported by `configure` (but you can use Tcl globbing to specify a range of triplets). If called with no arguments or an empty pattern, this procedure returns the build system triplet.

isbuild *`pattern`*

isremote Procedure

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

isremote *`board`*

This procedure is to be used instead of `is_remote`.

is_remote Procedure

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

is_remote *`board`*

Note that this procedure is now deprecated. Use `isremote` instead.

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

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*. *pattern* must be a full three-part configure triplet; in particular, you may not use the shorter aliases supported by `configure` (but you can use Tcl globbing to specify a range of triplets). If called with no arguments or an empty pattern, this procedure returns the host triplet.

ishost *`pattern`*

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*. *pattern* must be a full three-part configure triplet; in particular, you may not use the shorter aliases supported by `configure` (but you can use Tcl globbing to specify a range of triplets). If called with no arguments or an empty pattern, this procedure returns the target triplet.

istarget *pattern*

isnative Procedure

This procedure returns *1* if the current configuration has the same host and target (ie. it is a native configuration). Otherwise it returns *0*.

isnative

log_and_exit Procedure

log_and_exit

This procedure writes out the end of the test log and terminates `runtest`.

log_summary Procedure

log_summary *args*

`args`

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 a range of triplets). 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.

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 `message` argument.

pass *message*

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 `message` argument.

fail *message*

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) and the `message` argument.

xpass *message*

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*

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.

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

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.

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.

note Procedure

Appends an informational message to the log file. `note` writes in the log files a message beginning with *NOTE*, appending the `message` argument. 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 *messsage*

untested Procedure

Declares a test was not run. `untested` writes in the log file a message beginning with *UNTESTED*, appending the `message` argument. 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*

unresolved Procedure

Declares a test to have an unresolved outcome. `unresolved` writes in the log file a message beginning with *UNRESOLVED*, appending the `message` argument. 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*

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 `message` argument.

unsupported *message*

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.

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 patterns are also permitted.

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.

verbose Procedure

Test cases can use this procedure to issue helpful messages depending on the number of `-v`/`--verbose` options passed on the command line to `runtest`. It prints *message* if the value of the number of `-v` options passed is greater than or equal to the *loglevel* argument. The default log level is 1.

verbose *`-log -x -n message loglevel`*

`-log` Always write *message* to the log file, even if it won't be printed on the console.

`-x` Log the *message* into an XML file.

`-n` Print *message* without a trailing newline.

`--` Use this option if *message* begins with '-'.

`message` The log messsage.

`loglevel` The specified log level. The default level is 1.

load_lib Procedure

`load_lib` 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
```

testsuite Procedure

The `testsuite` procedure is a multiplex call for retrieving or providing information about the current testsuite.

testsuite file

The `testsuite file` command returns an absolute file name specified relative to either the testsuite source or object trees.

testsuite file ?-source|-object? -top|-test ?-hypothetical? ?--? *name...*

Any number of *names* are accepted and combined as if by `file join` with a directory relevant to the testsuite prepended.

`-object` Return a file name in the object tree.

`-source` Return a file name in the source tree.

`-top` Prepend the `testsuite` directory itself.

`-test` Prepend the directory containing the current test script.

`-hypothetical`
Allow the returned value to imply directories that do not exist.

`--` Use this option if the first *name* could begin with '`-`'.

One of `-top` or `-test` must be given; an error is raised otherwise.

Unless the `-hypothetical` option is given, any directories implied by the returned value will exist upon return. Implied directories are created in the object tree if needed. An error is raised if an implied directory does not exist in the source tree.

testsuite can call

The `testsuite can call` command is a feature test and returns a boolean value indicating if a subcommand under a multiplex point is available. This API call is needed because only the multiplex points themselves are visible to the Tcl info command.

testsuite can call *feature...*

Any number of words are joined together into a single name, beginning with a multiplex entry point and forming the full name of an API call as documented in this manual.

testcase Procedure

The `testcase` procedure is a multiplex call for retrieving or providing information about the state of the testing process.

testcase group

The `testcase group` command provides support for grouping tests into hierarchical groups within a test script.

Group names are internally tracked as Tcl lists, but are reported as strings delimited using forward slash ('`/`') characters. Individual name elements may not contain whitespace, but may contain forward slash. A group entered by traversing intermediate levels must be left by traversing those same levels. Groups must properly nest.

There are three uses:

testcase group

Return the current group as a string delimited with forward slash ('`/`') characters.

testcase group begin *name*

testcase group end *name*

These forms allow a group to be explicitly entered and left. The *name* parameter must be identical across a pair of these calls, and both the `begin` and `end` calls must be in the same file.

testcase group eval *name {code}*

This form is available to wrap the `begin` and `end` calls around the execution of the provided *code*. This form is preferred for convenience in top-level scripts, but the `begin` and `end` calls are preferred in helper procedures for performance.

A.2 Procedures For Remote Communication

The file `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 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.

call_remote Procedure

A standard procedure to call the appropriate *proc*. This proceure first looks for a board-specific version, then a protocol-specific version, and finally `call_remote` will call `standard_$proc`.

call_remote *type proc dest args*

`proc`

`dest`

`args`

check_for_board_status Procedure

This procedure inspected the named variable within the calling procedure for the expected output from the status wrapper. A non-negative value is returned if it exists. Otherwise, it returns -1. The output from the status wrapper is removed from the variable.

check_for_board_status *variable*

`variable` The name of the variable to check in the calling procedure. Be sure to pass the name of the variable (`var`) and not the value of the variable (`$var`).

file_on_build Procedure

file_on_build *op file args*

`op`

`file`

`args`

file_on_host Procedure

file_on_host *op file args*

`op`

`file`

`args`

local_exec Procedure

Run the specified command on the local machine, redirecting input from file `inp` (if non-empty), redirecting output to file `outp` (if non-empty), and waiting `timeout` seconds for the command to complete before killing it. A two-element list is returned: the exit status of the command and any output produced by the command. If output is redirected, this may or may not be empty. If output is redirected, both stdout and stderr will appear in the specified file.

local_exec *commandline inp outp timeout*

`inp` Redirect input into the input filename if not set to `""`.

`outp` Redirect output into the output filename if not set to `""`.

`timeout` Timeout in seconds.

remote_binary Procedure

This procedure sets the connection into *binary* mode. That is, there is no processing of input characters.

remote_binary *host*

`host` The host on which to set a binary connection.

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.

remote_download Procedure

Download a file to a destination machine. This procedure returns either an empty string (indicating failure) or the name of the file on the destination macine.

remote_download *dest file args*

`dest` Destination machine name.

`file` Filename.

`args` If the optional destination filename is specified, that filename will be used on the destination machine.

remote_exec Procedure

Execute the supplied program on a remote host. A two-element list is returned. The first element is the exit status of the program or -1 if execution failed. The second element is any output produced by the program. This may be an empty string if output from the program was redirected.

remote_exec *hostname program ?options? ?input? ?output? ?timeout?*

`hostname` Name of the host to execute the command on.

`program` Command to execute.

`options` Arguments to pass to the program.

`input` Input filename to feed to standard input of the command.

`output` Output filename where the output from the command should be written.

`timeout` Timeout value in seconds.

All of the optional positional arguments accept an empty string as a neutral value.

remote_expect Procedure

remote_expect *board timeout args*

`board`

`timeout`

`args`

remote_file Procedure

remote_file *dest args*

`dest`

`args`

remote_ld Procedure

remote_ld *dest prog*

`dest`

`prog`

remote_load Procedure

remote_load *dest prog args*

`dest`

`prog`

`args`

remote_open Procedure

Open connection to a remote host or target. This requires the `target_info` array be filled in with the proper information to work. It returns the spawn id of the process that is the connection.

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.

remote_pop_conn Procedure

Pop a previously-pushed connection from the stack. You should have closed the current connection before calling this procedure. Returns `pass` or `fail`.

remote_pop_conn *host*

`host`

remote_push_conn Procedure

Pushes the current connection onto a stack. Returns `pass` or `fail`.

remote_push_conn *host*

`host`

remote_raw_binary Procedure

remote_raw_binary *host*

`host`

remote_raw_close Procedure

remote_raw_close *host*

`host`

remote_raw_file Procedure

remote_raw_file *dest args*

`dest`

`args`

remote_raw_ld Procedure

remote_raw_ld *dest prog*

`dest`

`prog`

remote_raw_load Procedure

remote_raw_load *dest prog args*

`dest`

`prog`

`args`

remote_raw_open Procedure

remote_raw_open *args*

`args`

remote_raw_send Procedure

remote_raw_send *dest string*

`dest`

`string`

remote_raw_spawn Procedure

remote_raw_spawn *dest commandline*

`dest`

`commandline`

remote_raw_transmit Procedure

remote_raw_transmit *dest file*

`dest`

`file`

remote_raw_wait Procedure

remote_raw_wait *dest timeout*

`dest`

`timeout`

remote_reboot Procedure

Reboot the host. The return value of this procedure depends on the 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`

remote_send Procedure

remote_send *dest string*

`dest`

`string`

remote_spawn Procedure

Start a command on the destination. By default it is not possible to redirect I/O. If the command is successfully started, a positive spawn ID is returned. If the spawn fails, a negative value will be returned. Once the command has started, you can interact with it using `remote_expect` and `remote_wait` procedures.

remote_spawn *`dest commandline args`*

`dest` The destination.

`commandline`
The command to execute.

`args` If the optional keyword `readonly` is specified, input to the command may be redirected.

remote_swap_conn Procedure

Swap the current connection with the topmost one on the stack. Returns `pass` or `fail`.

remote_swap_conn *`host`*

remote_transmit Procedure

remote_transmit *`dest file`*

`dest`

`file`

remote_upload Procedure

remote_upload *`dest srcfile arg`*

`dest`

`srcfile`

`arg`

remote_wait Procedure

Wait for the last spawned command on the destination to complete. A list of two values is returned: the exit status (-1 if the program timed out) and any output produced by the command.

remote_wait *`dest timeout`*

`dest` The destination board.

`timeout` The timeout in seconds.

standard_close Procedure

This procedure closes a connection.

standard_close *`host`*

`host` The host to close the connection to.

standard_download Procedure

Downloads a file to a destination. It returns either the empty string (indicating failure) or the name of the file on the destination.

standard_download *dest file destfile*

`dest` Destination board.

`file` The name of the file to download.

`destfile` If the optional *destile* is specified, that filename will be used on the destination board.

standard_exec Procedure

standard_exec *hostname args*

`hostname`

`args`

standard_file Procedure

standard_file *dest op args*

standard_load Procedure

standard_load *dest prog args*

`dest`

`prog`

`args`

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`

standard_send Procedure

standard_send *dest string*

`dest`

`string`

standard_spawn Procedure

standard_spawn *dest commandline*

`dest`

`commandline`

standard_transmit Procedure

The default transmit procedure if none other exists. This feeds the file directly into the connection.

standard_transmit *dest file*

`dest`

`file` File to transmit.

standard_upload Procedure

standard_upload *dest srcfile destfile*

`dest`

`srcfile`

`destfile`

standard_wait Procedure

standard_wait *dest timeout*

`dest`

`timeout`

unix_clean_filename Procedure

This procedure returns an absolute version of the filename argument with '.' and '..' removed.

unix_clean_filename *dest file*

`dest`

`file` The filename.

A.3 Procedures For Using Utilities to Connect

kermit_open Procedure

kermit_open *dest args*

`dest`

`args`

kermit_command Procedure

kermit_command *dest args*

`dest`

`args`

kermit_send Procedure

kermit_send *dest string args*

`dest`

`string`

`args`

kermit_transmit Procedure

kermit_transmit *dest file args*

`dest`

`file`

`args`

telnet_open Procedure

This procedure opens a connection to a remote host using TELNET. This procedure sets the `fileid` field in the `board_info` array and returns the spawn id (or -1 for error).

telnet_open *hostname args*

`hostname` The host to connect to with TELNET.

`args` A list of options. Currently the only supported option is `raw`.

telnet_binary Procedure

Puts an existing TELNET connection into binary mode.

telnet_binary *hostname*

`hostname` Hostname for the connection.

tip_open Procedure

Connect to a host using `tip(1)`. This procedure sets the board `fileid` field with the `spawn_id` on success and, otherwise, returns -1.

tip_open *hostname*

`hostname` Hostname to connect to.

rlogin_open Procedure

rlogin_open *arg*

`arg`

rlogin_spawn Procedure

rlogin_spawn *dest cmdline*

`dest`

`cmdline`

rsh_open Procedure

rsh_open *hostname*

`hostname`

rsh_download Procedure

rsh_download *desthost srcfile destfile*

`desthost`

`srcfile`

`destfile`

rsh_upload Procedure

rsh_upload *desthost srcfile destfile*

`desthost`

`srcfile`

`destfile`

rsh_exec Procedure

rsh_exec *boardname cmd args*

`boardname`
`cmd`

`args`

ssh_close procedure

ssh_close *desthost*

`desthost`

ssh_exec procedure

ssh_exec *boardname program pargs inp outp*

`boardname`
`program`

`pargs`

`inp`

`outp`

ssh_download procedure

ssh_download *desthost srcfile destfile*

`desthost`

`srcfile`

`destfile`

ssh_upload procedure

ssh_upload *desthost srcfile destfile*

`desthost`

`srcfile`

`destfile`

ftp_open Procedure

Open an FTP connection.

ftp_open *host*

`host` The host to open the FTP connection to.

ftp_upload Procedure

Fetches a file from a remote host using FTP.

ftp_upload *host remotefile localfile*

`host` The host to transfer the file from.

`remotefile`
The filename at the remote end.

`localfile`
The filename to store locally.

ftp_download Procedure

Sends a file to a remote host using FTP.

ftp_download *host localfile remotefile*

`host` The host to transfer the file from.

`localfile`
The filename on the local system.

`remotefile`
The filename at the remote end.

ftp_close Procedure

Closes the FTP connection to a host.

ftp_close *host*

`host` The host connection to close.

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.

A.4 Procedures For Target Boards

default_link Procedure

default_link *board objects destfile flags*

This is the internal implementation for the [target_link procedure], page 61, and should not be directly called from testsuite code.

default_target_assemble Procedure

default_target_assemble *source destfile flags*

This is the internal implementation for the [target_assemble procedure], page 58, and should not be directly called from testsuite code.

default_target_compile Procedure

default_target_compile *source destfile type options*

This is the default implementation for the [target_compile procedure], page 58, and is used if the current target board does not have a special procedure for this purpose. See [target_compile procedure], page 58, for API details. Calling this procedure directly from testsuite code is deprecated.

pop_config Procedure

pop_config *type*

`type`

prune_warnings Procedure

prune_warnings *text*

`text`

push_build Procedure

push_build *name*

`name`

push_config Procedure

push_config *type name*

`type`

`name`

reboot_target Procedure

Reboot the target.

reboot_target

target_assemble Procedure

target_assemble *source destfile flags*

`source`

`destfile`

`flags`

target_compile Procedure

target_compile *source destfile type options*

`source` Source file or other arguments if *type* is `none`.

`destfile` Destination file or empty string to request output as return value.

`type` Type of output that should be produced.

`none`	Special applications where no source is actually given.
`preprocess`	Run the source files through the C preprocessor.
`assembly`	Produce assembler source from the compiler.
`object`	Produce binary object files.
`executable`	Produce an executable program.

`options` List of additional options:

Language-selection options:

`ada` Use an Ada compiler.

`c++` Use a C++ compiler.

`d` Use a compiler for the D language.

`f77` Use a compiler for Fortran 77.

`f90` Use a compiler for Fortran 90.

`go` Use a compiler for Go.

`rust` Use a compiler for Rust.

If none of these options are given, the C compiler is used by default. Giving multiple language-selection options is an error.

The `f77` option generally selects the `g77` compiler, while the `f90` option selects the newer `gfortran` frontend. Both of these can compile Fortran 77, but only `gfortran` supports Fortran 90.

Search path options:

`incdir=`*dir*
: Additional directory to search for preprocessor include files. Multiple uses of this option add multiple directories to the search path.

`libdir=`*dir*
: Additional directory to search for libraries. Multiple uses of this option add multiple directories to the search path.

Target options:

`debug` Compile with debugging information. Multiple uses of this option are treated as a single use.

`dest=`*`target`*
: Override the current target and compile for *target* instead. If this option is given multiple times, only the last use is significant.

`compiler=`*`command`*
: Override the defaults and use *command* as the compiler. If this option is given multiple times, only the last use is significant.

`linker=`*`command`*
: Override the defaults and use *command* to build executables. If this option is given multiple times, only the last use is significant.

`early_flags=`*`flags`*
: Prepend *flags* to the set of arguments to be passed to the compiler. Multiple uses of this option specify additional arguments.

`additional_flags=`*`flags`*
: Add *flags* to the set of arguments to be passed to the compiler. Multiple uses of this option specify additional arguments.

`optimize=`*`flags`*
: Specify optimization flags to be passed to the compiler. Nothing enforces that the flags given with option must actually be related to optimization, however. If this option is given multiple times, only the last use is significant.

`ldflags=`*`flags`*
: Add *flags* to the set of arguments to be passed to the linker. Note that these are passed literally to the compiler driver, without adding a special prefix to each option. If a '`-Wl,`' prefix is needed with GCC, it must be included in the given *flags*. As a group, the linker flags are only used if an executable is requested and are given special treatment with some languages. Multiple uses of this option specify additional arguments.

`ldscript=`*`script`*
: Specify a linker script, or more precisely, the argument to pass to the linker via the compiler driver to select a linker script. The *script* value is passed literally to the compiler driver. If this option is given multiple times, only the last use is significant.

`libs=`*`libs`*
: Specify additional libraries to be included in the link. The *libs* value is a space-separated list of libraries to include. Each element is checked, and if a file exists with that exact name, it is added to the list of sources to be given to the compiler. Otherwise, the element is passed literally to the compiler driver after any linker

flags specified with the `ldflags` option. Multiple uses of this option specify additional lists, which are concatenated in the order they are given.

Execution options:

`timeout=`*`timeout`*
: Abort the compile job if it is still running after *timeout* seconds. This is intended for compiler tests that are known to cause infinite loops upon failure.

`redirect=`*`file`*
: Instead of returning output emitted on `stdout`, place it into *file*.

The `target_compile` procedure also uses several global Tcl variables as overrides:

`CFLAGS_FOR_TARGET`
: If `CFLAGS_FOR_TARGET` is set, its value is prepended to the flags otherwise prepared for the compiler, even ahead of any board-specific flags inserted as a result of a language-selection option.

`LDFLAGS_FOR_TARGET`
: If `LDFLAGS_FOR_TARGET` is set, the set of arguments to be passed to linker is initialized to its value instead of an empty list. The `ldflags` option appends to this list.

`CC_FOR_TARGET`
: Override default compiler. If no other compiler is given and this variable is set, its value will be used instead of searching for a compiler or using the default from the target board configuration. The `compiler` option overrides this variable.

`CXX_FOR_TARGET`
: Override C++ compiler. If the `c++` option is given, this compiler will be used and the `compiler` option ignored.

`D_FOR_TARGET`
: Override D language compiler. If the `d` option is given, this compiler will be used and the `compiler` option ignored.

`F77_FOR_TARGET`
: Override Fortran 77 compiler. If the `f77` option is given, this compiler will be used and the `compiler` option ignored.

`F90_FOR_TARGET`
: Override Fortran 90 compiler. If the `f90` option is given, this compiler will be used and the `compiler` option ignored.

`GO_FOR_TARGET`
: Override Go compiler. If the `go` option is given, this compiler will be used and the `compiler` option ignored.

`GO_LD_FOR_TARGET`
: Override Go linker. If the `go` option is given, this linker will be used.

`RUSTC_FOR_TARGET`
: Override Rust compiler. If the `rust` option is given, this compiler will be used and the `compiler` option ignored.

`GNATMAKE_FOR_TARGET`
: Override Ada compiler. If the `ada` option is given, this compiler will be used and the `compiler` option ignored.

The `target_compile` procedure obtains most defaults from the target board configuration, but additionally inserts any flags specified as `cflags_for_target` on the *host* board configuration. If no host is set, the `unix` board configuration is checked for a `cflags_for_target` key. If the `cflags_for_target` key exists, its value is inserted into the set of arguments given to the compiler after any arguments given with the `additional_flags` option.

In DejaGnu 1.6.2 and older, this mechanism did not work reliably and the `unix` board configuration was always searched for the `cflags_for_target` key, regardless of the host board selected.

Also in DejaGnu 1.6.2 and older, the `dest` option interacted very badly with the language-selection options. There was no correct way to combine these options because the language-specific defaults would be read from the current target board configuration instead of the board configuration specified with the `dest` option. The closest solution was to always specify the language-selection option first, but this results in defaults appropriate for the current target, instead of the target selected with the `dest` option.

target_link Procedure

target_link *`objects destfile flags`*

`objects`

`destfile`

`flags`

A.5 Target Database Procedures

board_info Procedure

Searches the `board_info` array for the specified information.

board_info *`machine op args`*

`machine`

`op`

`args`

host_info Procedure

host_info *`op args`*

`op`

`args`

set_board_info Procedure

This checks if the `board_info` array entry has been set already and, if not, sets it to given value.

set_board_info *`entry value`*

`entry` Field of the `board_info` to set.

`value` Value to set the field to.

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.

set_currtarget_info Procedure

set_currtarget_info *`entry value`*

`entry`

`value`

target_info Procedure

target_info *`op args`*

`op`

`args`

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.

unset_currtarget_info Procedure

unset_currtarget_info *`entry`*

`entry`

push_target Procedure

This makes the target named *name* be the current target connection.

push_target *`name`*

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

pop_target Procedure

This unsets the current target connection.

pop_target

push_host Procedure

This procedure makes the host named *name* be the current remote host connection.

push_host `name`

`name` Name of the host to make the current connection.

pop_host Procedure

This unsets the current host connection.

pop_host

A.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.

${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

${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

${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

${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

A.7 Utility Procedures

getdirs Procedure

Returns a list of all the subdirectories in a single directory that match a glob pattern. If no directories match the pattern, then an empty list is returned.

This procedure is specialized as a search for tests in testsuites: `getdirs` ignores directories named '`testsuite`', '`config`', or '`lib`', and also ignores directories associated with a few revision control systems, specifically Git ('`.git`'), Subversion ('`.svn`'), CVS ('`CVS`'), RCS ('`RCS`'), and SCCS ('`SCCS`'). These ignored directories will not appear in the returned list, nor will they be examined in a recursive search.

getdirs *`-all rootdir pattern`*

`-all` If this option is given, then subdirectories will be matched recursively.

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

`pattern` The Tcl glob pattern to match. If you do not specify `pattern`, `getdirs` uses a default pattern of `*`.

relative_filename Procedure

Return a relative file name, given a starting point.

relative_filename *`base destination`*

`base` The starting point for relative file name traversal.

`destination`
The absolute file name that should be reached by appending the return value to *base*.

find Procedure

Search for files whose names match a glob pattern. Search subdirectories recursively, starting at a particular root directory. The result is the list of files whose names match. Filenames in the result include all intervening subdirectory names. If no files match the pattern, then an empty string is returned.

find *rootdir pattern*

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

`pattern` A glob pattern representing the files to find.

which Procedure

Searches the execution path for an executable file like the BSD `which(1)` utility. This procedure uses the shell environment variable `PATH`. It returns **0** if the binary is not in the path or if the `PATH` environment variable is not set. If the file is in the path, this procedure returns the full path to the file.

which *file*

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

grep Procedure

Search a named file for lines that contain a match for a regular expression. The result is a list of all the lines that match. If no lines match, the result is an empty string. All of the Tcl regular expression syntax is supported.

grep *-n filename regexp* **line**

`-n` The `-n` option prefixes matched lines in the result with the line number, just like GNU `grep` does. This option should be used in preference to the `line` keyword documented below.

`filename` The file to search.

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

`line` Use the optional keyword `line` to prefix matched lines in the result with the line number. This usage is deprecated.

prune Procedure

This procedure is deprecated and will be removed in a future 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.

runtest_file_p Procedure

Search *runtests* for *testcase* and return **1** if found, **0** if not. This is used by tools like compilers where each testcase is a file.

runtest_file_p *runtests testcase*

`runtests`

`runtests` is a list of two elements. The second is a copy of what was on the right side of the `=` if `foo.exp="..."` was specified, or an empty string if no such argument is present.

`testcase` The filename of the current testcase under consideration.

diff Procedure

Compares two files and returns **1** if they match (no differences) or **0** if not. If `verbose` is set, then it will print the differences to the console.

diff *`file1 file2`*

`file1` First file for the comparison.

`file2` Second file for the comparison.

setenv Procedure

Set an environment variable.

setenv *`var val`*

`var` The environment variable to set.

`val` The value to set the variable to.

unsetenv Procedure

Unset an environment variable.

unsetenv *`var`*

`var` The environment variable to unset.

getenv Procedure

Returns the value of the envrionment variable *var* if it is defined, otherwise an empty string is returned.

getenv *`var`*

`var` Environment variable to retrieve.

A.8 Libgloss, a free board support package (BSP)

Libgloss is a free board support package *BSP* commonly used with GCC and G`++` to produce a fully linked executable image for an embedded systems.

libgloss_link_flags Procedure

Finds the pieces of `libgloss` needed to link a set of object files into an executable. This usually means setting the `-L` and `-B` paths correctly.

libgloss_link_flags *`args`*

`args` Ignored.

libgloss_include_flags Procedure

This procedure always returns an empty string. It is provided for consistency.

libgloss_include_flags *`args`*

`args` Ignored.

newlib_link_flags Procedure

Return the options needed to link an executable with `newlib`. This usually means setting the `-L` and `-B` paths correctly.

newlib_link_flags *args*

`args` Ignored.

newlib_include_flags Procedure

Return the options needed to locate the `newlib` header files.

newlib_include_flags *args*

`args` Ignored.

libio_include_flags Procedure

libio_include_flags *args*

Return the options needed to locate the `libio` header files.

`args` Ignored.

libio_link_flags Procedure

libio_link_flags *args*

Return the options needed to link an executable with `libio`. This usually means setting the `-L` and `-B` paths correctly.

`args` Ignored.

g++_include_flags Procedure

Return the options needed to locate the C++ stnadard library header files.

g++_include_flags *args*

`args` Ignored.

g++_link_flags Procedure

g++_link_flags *args*

Return the options needed to link an executable with `libg++`. This usually means setting the `-L` and `-B` paths correctly.

`args` Ignored.

libstdc++_include_flags Procedure

libstdc++_include_flags *args*

Return the options needed to locate the C++ stnadard library header files.

`args` Ignored.

libstdc++_link_flags Procedure

libstdc++_link_flags *args*

`args`

get_multilibs Procedure

get_multilibs *args*

`args`

find_binutils_prog Procedure

find_binutils_prog *name*

`name`

find_gcc Procedure

Looks for a copy of the GNU C compiler in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_gcc

find_gcj Procedure

Looks for a copy of the GNU Java compiler in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_gcj

find_g++ Procedure

Looks for a copy of the GNU C++ compiler in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_g++

find_g77 Procedure

Looks for a copy of the GNU Fortran 77 compiler in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_g77

find_gfortran Procedure

Looks for a copy of the GNU Fortran compiler in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_gfortran

find_go Procedure

Looks for a copy of the GNU compiler for the Go language in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_go

find_go_linker Procedure

Looks for a copy of the special linker associated with the GNU compiler for the Go language in the build tree and in the `PATH`. This will also return the proper transformed name for a cross-compiler if the build tree is configured for one.

find_go_linker

find_rustc Procedure

Looks for a copy of a compiler for the Rust language in the build tree and in the `PATH`. The Rust compiler is different and this procedure also ensures that it will be called with options to suppress output coloration.

find_rustc

process_multilib_options Procedure

process_multilib_options *args*

`args`

add_multilib_option Procedure

add_multilib_option *args*

`args`

find_gas Procedure

find_gas

find_ld Procedure

find_ld

build_wrapper Procedure

build_wrapper *gluefile*

`gluefile`

winsup_include_flags Procedure

winsup_include_flags *args*

`args`

winsup_link_flags Procedure

winsup_link_flags *args*

`args`

A.9 Procedures for debugging your scripts

bt Procedure

This procedure prints a backtrace using the `w` command from the Tcl debugger.

bt

dumpvars Procedure

This procedure prints the values of the global variables that match a glob pattern. Abbreviation: *dv*.

dumpvars *pattern*

`pattern` The global variables to dump.

dumplocals Procedure

This procedure prints the values of local variables that match a glob pattern. Abbreviation: *dl*.

dumplocals *pattern*

`pattern` The local variables to dump.

dumprocs Procedure

This procedure dumps the body of all procs that match a glob pattern. It is abbreviated as *dp*.

dumprocs *pattern*

`pattern` The proc bodies to dump.

dumpwatch Procedure

This procedure prints all of the watchpoints matching a glob pattern. It is abbreviated as *dw*.

dumpwatch *pattern*

`pattern` The watchpoints to dump.

watcharray Procedure

watcharray *array element type*

`array`

`element`

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

watchvar Procedure

watchvar *var type*

`var`

`type`

watchunset Procedure

This breaks program execution when the variable `var` is unset. Abbreviation: *wu*.

watchunset *pattern*

`pattern`

watchwrite Procedure

This breaks program execution when the variable `var` is written. Abbreviation: *ww*.

watchwrite *var*

`var` The variable to watch.

watchread Procedure

This breaks program execution when the variable `var` is read. Abbreviation: *wr*.

watchread *var*

`var` The variable to watch.

watchdel Procedure

This deletes a watchpoint from the watch list. Abbreviation: *wd*.

watchdel *pattern*

`pattern`

print Procedure

This prints the value of a variable. Abbreviation: *p*.

print *var*

`var` The variable to print.

quit Procedure

This makes `runtest` exit. Abbreviation: *q*.

quit

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Concept Index

Procedure Index

V

W

X

Variable Index

B

C

E

P

S

T