DejaGnu

The GNU Testing Framework

Rob Savoye
Free Software Foundation

DejaGnu: The GNU Testing Framework
by Rob Savoye

1.4.2 Edition

Revision History

Revision 0.6.1	2001-2-16	Revised by: rob@welcomehome.org
Add info on the new dejagnu.h file.		
Revision 0.6	2001-2-16	Revised by: rob@welcomehome.org
Updated for new release.		
Revision 0.5	2000-1-24	Revised by: rob@welcomehome.org
Initial version after conversion to DocBook.		

Table of Contents

Abstract

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

DejaGnu itself is merely a framework for creation of a test suites. Test suites are distributed separately for each GNU tool.

Chapter 1. Overview

What is DejaGnu ?

DejaGnu is a framework for testing other programs. Its purpose is to provide a single front end for all tests. Think of it as a custom library of Tcl procedures crafted to support writing a test harness. A *Test Harness* is the testing infrastructure that is created to support a specific program or tool. Each program can have multiple test suites, all supported by a single test harness. DejaGnu is written in Expect, which in turn uses Tcl -- Tool command language. There is more information on Tcl at the Scriptics[1] web site, and the Expect web site is at NIST[2].

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 Unix based host) on any target architecture that DejaGnu supports. Currently DejaGnu runs tests on many single board computers, whose operating software ranges from just a boot monitor to a full-fledged, Unix-like realtime 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 test suites. 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 test suites themselves. Tests are usually written in Expect using Tcl, but you can also use a Tcl script to run a test suite that is not based on Expect. (expect script filenames conventionally use *.exp* as a suffix; for example, the main implementation of the DejaGnu test driver is in the file runtest.exp.)

Julia Menapace first coined the term "Deja Gnu" to describe an earlier testing framework at Cygnus Support she had written for **GDB**. When we replaced it with the Expect-based framework, it was like DejaGnu all over again... But more importantly, it was also named after my daughter,Deja Snow Savoye[3] (now 9 years old in Dec of 1998), who was a toddler during DejaGnu's creation.

What's New In This Release

This release has a number of substantial changes over version 1.3. The most visible change is that the version of Expect and Tcl included in the release are up-to-date with the current stable net releases. The biggest change is years of modifications to the target configuration system, used for cross testing. While this greatly improved cross testing, is has made that subsystem very complicated. The goal is to have this entirely rewritten using iTcl by the next release. Other changes are:

- More builtin support for building target binaries with the correct linker flags. Currently this only works with GCC as the cross compiler, preferably with a target supported by Libgloss.
- Lots of little bug fixes from years of heavy use at Cygnus Solutions.
- DejaGnu now uses Automake for Makefile configuration.
- Updated documentation, now in SGML (using the free GNU DocBook tools[4]) format.

- NT support. There is beta level support for NT that is still a work in progress. This requires the Cygwin[5] POSIX system for NT.

NT Support

To use DejaGnu on NT, you need to first install the Cygwin[6] release. This works as of the B20.1 release. Cygwin is a POSIX system for NT. This covers both utility programs, and a libray that adds POSIX system calls to NT. Among them is pseudo tty support for NT that emulates the POSIX pty standard. The latest Cygwin is always available from this location[7]. This works well enough to run *"make check"* of the GNU development tree on NT after a native build. But the nature of pty's on NT is still evolving. Your mileage may vary...

Design Goals

DejaGnu grew out of the internal needs of Cygnus Solutions. (then Cygnus Support). Cygnus maintained and enhanced a variety of free programs in many different environments, and we 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 cross-development testing.
- permited testing interactive programs, like **GDB**; and
- permited testing batch oriented programs, like **GCC**.

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

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

A POSIX conforming test framework

DejaGnu conforms to the POSIX 1003.3 standard for test frameworks. I was also a member of that committe.

The POSIX standard 1003.3 defines what a testing framework needs to provide, in order to permit the creation of POSIX conformance test suites. This standard is primarily oriented to running POSIX conformance tests, but its requirements also support testing of features not related to POSIX conformance. POSIX 1003.3 does not specify a particular testing framework, but at this time there is only one other POSIX conforming test framework: TET. 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 daytime or nighttime. It is important to note that the standard being tested is never 1003.3; the standard being tested is some other standard, for which the assertions were written.

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

PASS

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

XFAIL

POSIX 1003.3 does not incorporate the notion of expected failures, so *PASS*, instead of *XPASS*, must also be returned for test cases which were expected to fail and did not. This means that *PASS* is in some sense more ambiguous than if *XPASS* is also used.

FAIL

A test has produced the bug it was intended to capture. That is, it has demonstrated that the assertion is false. The *FAIL* message is based on the test case only. Other messages are used to indicate a failure of the framework. As with *PASS*, POSIX tests must return *FAIL* rather than *XFAIL* even if a failure was expected.

UNRESOLVED

A test produced indeterminate results. Usually, this means the test executed in an unexpected fashion; this outcome requires that a human being go over results, to determine if the test should have passed or failed. This message is also used for any test that requires human intervention because it is beyond the abilities of the testing framework. Any unresolved test should resolved to *PASS* or *FAIL* before a test run can be considered finished.

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

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

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

UNTESTED

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

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

UNSUPPORTED

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

DejaGnu uses the same output procedures to produce these messages for all test suites, and these procedures are already known to conform to POSIX 1003.3. For a DejaGnu test suite to conform to POSIX 1003.3, you must avoid the *setup*xfail} procedure as described in the *PASS* section above, and you must be careful to return *UNRESOLVED* where appropriate, as described in the *UNRESOLVED* section above.

Notes

1. http://www.scriptics.com
2. http://expect.nist.gov
3. mailto:deja@welcomehome.org
4. http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html
5. http://sources.redhat.com
6. http://sources.redhat.com/cygwin
7. http://sources.redhat.com/cygwin

Chapter 2. Running Tests

There are two ways to execute a test suite. The most common way is when there is existing support in the `Makefile`. This support consists of a *check* target. The other way is to execute the **runtest** program directly. To run **runtest** directcly from the command line requires either all the correct options, or the Local Config File must be setup correctly.

Make check

To run tests from an existing collection, first use **configure** as usual to set up the build directory. Then try typing:

```
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 builds is the *site.exp*. The site.exp file contains various variables that DejaGnu used to dertermine the configuration of the program being tested. This is mostly for supporting remote testing.

The *check* target is supported by GNU Automake. To have DejaGnu support added to your generated `Makefile.in`, just add the keyword dejagnu to the AUTOMAKE_OPTIONS variable in your `Makefile.am` file.

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`.

Runtest

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

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

Output States

`runtest` flags the outcome of each test as one of these cases. A POSIX Conforming Test Framework for a discussion of how POSIX specifies the meanings of these cases.

PASS

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

XPASS

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

FAIL

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

XFAIL

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

UNRESOLVED

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

UNTESTED

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

UNSUPPORTED

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

runtest may also display the following messages:

ERROR

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

WARNING

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

NOTE

An informational message about the test case.

Invoking Runtest

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

`--all` (-a)

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 [string]`

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

`--host [string]`

string is a full configuration "triple" name as used by *configure*. Use this option to override the default string recorded by your configuration's choice of host. This choice does not change how anything is actually configured unless --build is also specified; it affects *only* DejaGnu procedures that compare the host string with particular values. The procedures *ishost*, *istarget*, *isnative*, and *setup*xfail} are affected by *--host*. In this usage, *host* refers to the machine that the tests are to be run on, which may not be the same as the *build* machine. If *--build* is also specified, then *--host* refers to the machine that the tests wil, be run on, not the machine DejaGnu is run on.

`--host_board [name]`

The host board to use.

`--target [string]`

Use this option to override the default setting (running native tests). *string* is a full configuration "triple" name of the form *cpu-vendor-os* as used by **configure**. This option changes the configuration *runtest* uses for the default tool names, and other setup information.

`--debug` (-de)

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

`--help` (-he)

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

`--ignore [name(s)]`

The names of specific tests to ignore.

`--objdir [path]`

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

`--outdir [path]`

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

`--reboot [name]`

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

`--srcdir [path]`

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

`--strace [number]`

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*.

`--connect [program]`

Connect to a target testing environment as specified by *type*, if the target is not the computer running *runtest*. For example, use *--connect* to change the program used to connect to a "bare board" boot monitor. The choices for *type* in the DejaGnu 1.4 distribution are *rlogin*, *telnet*, *rsh*, *tip*, *kermit*, and *mondfe*.

The default for this option depends on the configuration most convenient communication method available, but often other alternatives work as well; you may find it useful to try alternative connect methods if you suspect a communication problem with your testing target.

`--baud [number]`

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

`--target_board [name(s)]`

The list of target boards to run tests on.

`--tool[name(s)]`

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

`--tool_exec [name]`

The path to the tool executable to test.

`--tool_opts [options]`

A list of additional options to pass to the tool.

`--verbose` (-v)

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

`--version` (-V)

Prints out the version numbers of DejaGnu, *expect* and Tcl, and exits without running any tests.

`--D[0-1]`

Start the internal Tcl debugger. The Tcl debugger supports breakpoints, single stepping, and other common debugging activities. See the document "Debugger for Tcl Applications" by Don Libes. (Distributed in PostScript form with *expect* as the file `expect/tcl-debug.ps`.. If you specify *-D1*, the *expect* shell stops at a breakpoint as soon as DejaGnu invokes it. If you specify *-D0*, DejaGnu starts as usual, but you can enter the debugger by sending an interrupt (e.g. by typing **C-c**).

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

Specify the names of testsuites to run. By default, *runtest* runs all tests for the tool, but you can restrict it to particular testsuites by giving the names of the *.exp expect* scripts that control them. *testsuite*.exp may not include path information; use plain filenames.

`testfile`.exp="testfile1 ..."

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

tclvar=value

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

The default Tcl variables used for most tools are defined in the main DejaGnu *Makefile*; their values are captured in the `site.exp` file.

Common Options

Typically, you don't need must to use any command-line options. `--tool` used is only required when there are more than one test suite 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:

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

Test output follows, ending with:

```
=== gdb Summary ===

# of expected passes 508
# of expected failures 103
/usr/latest/bin/gdb version 4.14.4 -nx
```

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

```
eg$ runtest--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 *--all* option. For more verbose output about processes being run, communication, and so on, use *--verbose*. To see even more output, use multiple *--verbose* options. for a more detailed explanation of each **runtest** option.

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

The files DejaGnu produces.

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

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

Summary File

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

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

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

Example 2-1. Here is a short sample summary log

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

Log File

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

Example 2-2. Here is a brief example showing a detailed log for G++ tests

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

               === g++ tests ===

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

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

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

               === g++ Summary ===

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

Debug Log File

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

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

Example 2-3. 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 2-4. Here is an excerpt from the debugging log for a GDB test:

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

Chapter 3. Customizing DejaGnu

The site configuration file, `site.exp`, captures configuration-dependent values and propagates them to the DejaGnu test environment using Tcl variables. This ties the DejaGnu test scripts into the **configure** and **make** programs. If this file is setup correctly, it is possible to execute a test suite merely by typing **runtest**.

DejaGnu supports two `site.exp` files. The multiple instances of `site.exp` are loaded in a fixed order built into DejaGnu. 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 *master* `site.exp`, capturing configuration values that apply to DejaGnu across the board, in each configuration-specific subdirectory of the DejaGnu library directory. **runtest** loads these values first. The master `site.exp` contains the default values for all targets and hosts supported by DejaGnu. This master file is identified by setting the environment variable DEJAGNU to the name of the file. This is also refered to as the "global" config file.

Any directory containing a configured test suite also has a local `site.exp`, capturing configuration values specific to the tool under test. Since **runtest** loads these values last, the individual test configuration can either rely on and use, or override, any of the global values from the global `site.exp` file.

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

You can also have a file in your home directory called `.dejagnurc`. This gets loaded first before the other config files. Usually this is used for personal stuff, like setting 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.

Local Config File

It is usually more convenient to keep these *manual overrides* in the `site.exp` local to each test directory, rather than in the global `site.exp` in the installed DejaGnu library. This file is mostly for supplying tool specific info that is required by the test suite.

All local `site.exp` files have two sections, separated by comment text. The first section is the part that is generated by **make**. It is essentially a collection of Tcl variable definitions based on `Makefile` environment variables. Since they are generated by **make**, they contain the values as specified by **configure**. (You can also customize these values by using the `--site` option to **configure**.) In particular, this section contains the `Makefile` variables for host and target configuration data. Do not edit this first section; if you do, your changes are replaced next time you run **make**.

Example 3-1. The first section starts with

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

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

Example 3-2. The first section ends with this line

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

You can make any changes under this line. If you wish to redefine a variable in the top section, then just put a duplicate value in this second section. Usually the values defined in this config file are related to the configuration of the test run. This is the ideal place to set the variables host_triplet, build_triplet, target_triplet. All other variables are tool dependant, 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.

Example 3-3. Local Config File

```
## these variables are automatically generated by make ##
# Do not edit here. If you wish to override these values
# add them to the last section
set rootme "/build/devo-builds/i586-pc-linux-gnulibc1/gcc"
set host_triplet i586-pc-linux-gnulibc1
set build_triplet i586-pc-linux-gnulibc1
set target_triplet i586-pc-linux-gnulibc1
set target_alias i586-pc-linux-gnulibc1
set CFLAGS ""
set CXXFLAGS "-I/build/devo-builds/i586-pc-linux-gnulibc1/gcc/../libio -I$srcdir/../libg++/src -I$srcdir/../libio -I$srcdir/../libstdc++ -I$srcdir/../libstdc++/stl -L/build/devo-builds/i586-pc-linux-gnulibc1/gcc/../libg++ -L/build/devo-builds/i586-pc-linux-gnulibc1/gcc/../libstdc++"
append LDFLAGS " -L/build/devo-builds/i586-pc-linux-gnulibc1/gcc/../ld"
set tmpdir /build/devo-builds/i586-pc-linux-gnulibc1/gcc/testsuite
set srcdir "${srcdir}/testsuite"
## All variables above are generated by configure. Do Not Edit ##
```

This file defines the required fields for a local config file, namely the three config triplets, and the srcdir. It also defines several other Tcl variables that are used exclusivly by the GCC test suite. 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.

Global Config File

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

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

Example 3-4. Global Config file

```
# Make sure we look in the right place for the board description files.
if ![info exists boards_dir] {
    set boards_dir {}
}
lappend boards_dir "/nfs/cygint/s1/cygnus/dejagnu/boards"
```

```
verbose "Global Config File: target_triplet is $target_triplet" 2
global target_list

case "$target_triplet" in {
    { "native" } {
        set target_list "unix"
    }
    { "sparc64-*elf" } {
        set target_list "sparc64-sim"
    }
    { "mips-*elf" } {
        set target_list "mips-sim wilma barney"
    }
    { "mips-lsi-elf" } {
        set target_list "mips-lsi-sim{,soft-float,el}"
    }
    { "sh-*hms" } {
        set target_list { "sh-hms-sim" "bloozy" }
    }
}
```

In this case, we have support for several cross compilers, that all run on this host. For testing on operating systems that don't support Expect, DejaGnu can be run on the local build machine, and it can connect to the remote host and run all the tests for this cross compiler on that host. All the remote OS requires is a working telnetd.

As you can see, all one does is set the variable target_list to the list of targets and options to test. The simple settings, like for *sparc64-elf* only require setting the name of the single board config file. The *mips-elf* target is more complicated. Here it sets the list to three target boards. One is the default mips target, and both *wilma barney* are symbolic names for other mips boards. Symbolic names are covered in the Adding A New Board chapter. The more complicated example is the one 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) option. Needless to say, this last feature is mostly compiler specific.

Board Config File

The board config file is where board specfic config data is stored. A board config file contains all the higher-level configuration settings. There is a rough inheritance scheme, where it is possible to base a new board description file on an existing one. There are also collections of custom procedures for common environments. For more information on adding a new board config file, go to the Adding A New Board chapter.

An example board config 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 in square brackets *[]* are *helper procedures*. Thes are used to find parts of a tool chain required to build an executable image that may reside in various locations. This is mostly of use for when the startup code, the standard C lobraries, or the tool chain itself is part of your build tree.

Example 3-5. Board Config 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 statuses 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. The first one, `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. The next helper procedures are `libgloss_include_flags` & `libgloss_link_flags`. These return the proper flags to compiler and link an executable image using Libgloss, the GNU BSP (Board Support Package). The final procedures are `newlib_include_flag` & `newlib_include_flag`. These find the Newlib C library, which is a reentrant standard C library for embedded systems comprising of non GPL'd code.

Remote Host Testing

Note: Thanks to Dj Delorie for the original paper that this section is based on.

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/Win3.1, MacOS, and win95/win98/NT.

The recommended source for a win95/win98/NT based ftp server is to get IIS (either IIS 1 or Personal Web Server) from 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 win95/win98/NT, go to the Ataman[2] web site, pick up the Ataman Remote Logon Services for Windows, and install it. You can get started on the eval period anyway. Add IUSR_FOOBAR to the list of allowed users, set the HOME directory to be the same as the FTP default directory. Change the Mode prompt to simple.

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

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

Example 3-6. 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:

Example 3-7. 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:

Example 3-8. 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:

Example 3-9. 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.

Example 3-10. 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):

Example 3-11. 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.

Config File Values

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

Command Line Option Variables

In the user editable second section of the Personal Config File you can not only override the configuration variables captured in the first section, but also specify default values for all on the **runtest** command line options. Save for `--debug`, `--help`, and `--version`, each command line option has an associated Tcl variable. Use the Tcl **set** command to specify a new default value (as for the configuration variables). The following table describes the correspondence between command line options and variables you can set in `site.exp`. Invoking Runtest, for explanations of the command-line options.

Table 3-1. Tcl Variables For Command Line Options

runtest	Tcl	option	variable	description
--all	all_flag	display all test results if set		
--baud	baud	set the default baud rate to something other than 9600.		
--connect	connectmode	**rlogin**, **telnet**, **rsh**, **kermit**, **tip**, or **mondfe**		
--outdir	outdir	directory for `tool.sum` and `tool.log`.		
--objdir	objdir	directory for pre-compiled binaries		

runtest	**Tcl**	**option**	**variable**	**description**
--reboot	reboot	reboot the target if set to *"1"*; do not reboot if set to *"0"* (the default).		
--srcdir	srcdir	directory of test subdirectories		
--strace	tracelevel	a number: Tcl trace depth		
--tool	tool	name of tool to test; identifies init, test subdir		
--verbose	verbose	verbosity level. As option, use multiple times; as variable, set a number, 0 or greater.		
--target	target_triplet	The canonical configuration string for the target.		
--host	host_triplet	The canonical configuration string for the host.		
--build	build_triplet	The canonical configuration string for the build host.		
--mail	address	Email the output log to the specified address.		

Personal Config File

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

Example 3-12. Personal Config File

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

Here I set all_flag so I see all the test cases that PASS along with the ones that FAIL. I also set RLOGIN to the BSD version. I have Kerberos installed, and when I rlogin to a target board, it usually isn't supported. So I use the non secure version rather than the default that's in my path. I also set RSH to the SSH secure shell, as rsh is mostly used to test unix machines within a local network here.

Notes

1. http://www.microsoft.com
2. http://ataman.com

Chapter 4. Extending DejaGnu

Adding A New Test Suite

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

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 test suites. Unfortunately, well-established test suites have a way of developing their own conventions: as test writers become more experienced with DejaGnu and with Tcl, they accumulate more utilities, and take advantage of more and more features of Expect and Tcl in general.

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

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

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

- 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.in for use with GNU Autoconf.

- Create `Makefile.in` (if you are using Autoconf), or `Makefile.am`(if you are 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 Make Check 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 Local Config File The check target must run the **runtest** program to execute 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.

Example 4-1. Sample Makefile.in Fragment

```
# 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‘

# The 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?
```

- Create a directory (in `testsuite`) called `config`. Make a *Tool Init File* in this directory. Its name must start with the target_abbrev value, or be named `default.exp` so call it `config/unix.exp` for our Unix based example. This is the file that contains the target-dependent procedures. Fortunately, on Unix, most of them do not have to do very much in order for **runtest** to run.

  If the program being tested is not interactive, you can get away with this minimal `unix.exp` to begin with:

Example 4-2. Simple Batch Program Tool Init File

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

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

Example 4-3. Simple Interactive Program Tool Init File

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

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

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

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

  Example 4-4. Testing A New Tool Config

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

- Back in the `testsuite` (top level) directory, run **configure**. Typically you do this while in the build directory. You may have to specify more of a path, if a suitable configure is not available in your execution path.
- e now ready to triumphantly type **make check** or **runtest**. You should see something like this:

  Example 4-5. Example Test Case Run

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

               === example tests ===

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

               === example Summary ===
  ```

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

- 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.

Adding A New Target

DejaGnu has some additional requirements for target support, beyond the general-purpose provisions of configure. 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. This permits DejaGnu itself to remain target independent.

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 @samp{--debug} option to see what is really going on.

When you code an initialization module, be generous in printing information controlled by the `verbose` procedure.

For cross targets, most of the work is in getting the communications right. Communications code (for several situations involving IP networks or serial lines) is available in a DejaGnu library file.

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.

Adding A New Board

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

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

Example 4-6. Testing a New Board Config File

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

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

Example 4-7. Example Board Config File

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

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

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

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

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

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

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

# No signals.
set_board_info gdb,nosignals 1

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

Board Config File Values

These fields are all in the board_info These are all set by using the `set_board_info` procedure. The parameters are the field name, followed by the value to set the field to.

Table 4-1. Common Board Info Fields

Field	Sample 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.

Field	Sample Value	Description
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.

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.

Table 4-2. 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_init_command	"set mipsfpu none"	
gdb,cannot_call_functions	1	Whether GDB can call functions on the target,
gdb,noargs	1	Whether the target can take command line arguments.

Field	Sample Value	Description
gdb,nosignals	1	Whether there are signals on the target.
gdb,short_int	1	
gdb,start_symbol	"_start";	The starting symbol in the executable.
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	"print/x \$fsr = 0x0"	
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.
use_vma_offset	1	
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	
nullstone,lib	"mips-clock.c"	
nullstone,ticks_per_sec	3782018	
sys_speed_value	200	
target_install	{sh-hms}	

Writing A Test Case

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

The GCC tests are a good example of batch oriented tests. All GCC tests consist primarily of a call to a single common procedure, Since all the tests either have no output, or only have a few warning messages when successfully compiled. Any non-warning output is a test failure. All the C code needed is kept in the test directory. The test driver, written in 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` in the GCC test suite. Most tests of this kind use very few expect features, and are coded almost purely in Tcl.

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

- 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.

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 variable also called *verbose*) to control how much output to generate. This will make it easier for other people running the test to debug it if necessary. Whenever possible, if *$verbose* is *0*, there should be no output other than the output from *pass*, *fail*, *error*, and *warning*. Then, to whatever extent is appropriate for the particular test, allow successively higher values of *$verbose* to generate more information. Be kind to other programmers who use your tests: provide for a lot of debugging information.
- Output from the internal debugging functions of Tcl and Expect. There is a command line options for each; both forms of debugging output are recorded in the file `dbg.log` in the current directory.

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

- Use `--strace` to see more detail at the Tcl level; this shows how Tcl procedure definitions expand, as they execute. The associated number controls the depth of definitions expanded.

- Finally, if the value of *verbose* is 3 or greater,DejaGnu turns on the expect command **log_user**. This command prints all expect actions to the expect standard output, to the detailed log file, and (if `--debug` is on) to `dbg.log`.

Adding A Test Case To A Test Suite.

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

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

```
runtest --tool
      gcc
```

.

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

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

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

Hints On Writing A Test Case

It is safest to write patterns that match all the output generated by the tested program; this is called closure. If a pattern does not match the entire output, any output that remains will be examined by the next **expect** command. In this situation, the precise boundary that determines which **expect** command sees what is very sensitive to timing between the Expect task and the task running the tested tool. As a result, the test may sometimes appear to work, but is likely to have unpredictable results. (This problem is particularly likely for interactive tools, but can also affect batch tools---especially for tests that take a long time to finish.) The best way to ensure closure is to use the `-re` option for the **expect** command to write the pattern as a full regular expressions; then you can match the end of output using a *$*. It is also a good idea to write patterns that match all available output by using *.** after the text of interest; this will also match any intervening blank lines. Sometimes an alternative is to match end of line using *\r* or *\n*, but this is usually too dependent on terminal settings.

Always escape punctuation, such as *(* or *"*, in your patterns; for example, write *\(*. If you forget to escape punctuation, you will usually see an error message like

```
extra
      characters after close-quote.
```

If you have trouble understanding why a pattern does not match the program output, try using the `--debug` option to **runtest**, and examine the debug log carefully.

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

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

Special variables used by test cases.

There are special variables used by test cases. These contain other information from DejaGnu. Your test cases can use these variables, with conventional meanings (as well as the variables saved in `site.exp`. You can use the value of these variables, but they should never be changed.

$prms_id

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

$item bug_id

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

$subdir

The subdirectory for the current test case.

$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.

$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.

Chapter 5. Unit Testing

What Is Unit Testing ?

Most regression testing as done by DejaGnu is system testing. This is 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 series, with the test case having to check private data or global variables to see if the function or method worked.

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

The dejagnu.h Header File

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

Chapter 6. Reference

Obtaining DejaGnu

You can obtain DejaGnu from the DejaGnu web site at the Free Software Foundation[1], which is at www.gnu.org/software/dejagnu/ [2]

Installation

Once you have the DejaGnu source unpacked and available, you must first configure the software to specify where it is to run (and the associated defaults); then you can proceed to installing it.

Configuring DejaGnu

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

```
../dejagnu-1.4.2/configure
```

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

You may also want to use the **configure** option *--prefix* to specify where you want DejaGnu and its supporting code installed. By default, installation is in subdirectories of `/usr/local`, but you can select any alternate directory altdir by including `--prefix`{altdir}} on the **configure** command line. (This value is captured in the Makefile variables *prefix* and *exec*prefix}.)

Save for a small number of example tests, the DejaGnu distribution itself does not include any test suites; these are available separately. Test suites for the GNU development tools are included in those releases. 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.

Installing DejaGnu

To install DejaGnu in your filesystem (either in `/usr/local`, or as specified by your *--prefix* option to *configure*), execute.

```
eg$ make install
```

*make install*does thes things for DejaGnu:

- Look in the path specified for executables $exec_prefix) for directories called `lib` and `bin`. If these directories do not exist, *make install* creates them.
- Create another directory in the `share` directory, called `dejagnu`, and copy all the library files into it.

- Create a directory in the `dejagnu/share` directory, called `config`, and copy all the configuration files into it.
- Copy the *runtest* shell script into `$exec_prefix/bin`.
- Copy `runtest.exp` into `$exec_prefix/lib/dejagnu`. This is the main Tcl code implementing DejaGnu.

Builtin Procedures

DejaGnu provides these Tcl procedures.

Core Internal Procedures

Mail_file Procedure

```
mail_file(file to subject);
```

Open_logs Procedure

```
open_logs();
```

Close_logs Procedure

```
close_logs();
```

Isbuild Procedure

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

```
isbuild(pattern);
```

`pattern`

Is_remote Procedure

```
is_remote(board);
```

is3way Procedure

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

```
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*. *host* must be a full three-part configure host name; in particular, you may not use the shorter nicknames supported by configure (but you can use wildcard characters, using shell syntax, to specify sets of names).

```
ishost(pattern);
```

Istarget Procedure

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

```
istarget(args);
```

Isnative Procedure

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

```
isnative();
```

Unknown Procedure

```
unknown(args);
```

args

Clone_output Procedure

```
clone_output(message);
```

message

Reset_vars Procedure

```
reset_vars();
```

Log_and_exit Procedure

```
log_and_exit();
```

Log_summary Procedure

```
log_summary(args);
```

args

Cleanup Procedure

```
cleanup();
```

Setup_xfail Procedure

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

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

> **Warning**
>
> Warning you must clear the expected failure after using setup_xfail in a test case. Any call to `pass` or `fail`l 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 it this test case to a bug tracking system.

Record_test Procedure

```
record_test(type message args);
```

type

message

args

Pass Procedure

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

```
pass(string);
```

string

The string to use for this 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 argument *string*.

```
fail(string);
```

string

The string to use for this FAIL message.

Xpass Procedure

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

```
xpass(string);
```

string

The string to use for this output state.

Xfail Procedure

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

```
xpass(string);
```

string

The string to use for this output state.

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(string number );
```

string

number

The optional number to set the error counter. Thius 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(string number );
```

string

number

The optional number to set the error counter. Thius 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 argument *string*. Use `note` sparingly. The `verbose` should be used for most such messages, but in cases where a message is needed in the log file regardless of the verbosity level use `note`.

```
note(string);
```

string

The string to use for this note.

Untested Procedure

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

```
untested(string);
```

string

The string to use for this output state.

Unresolved Procedure

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

```
unresolved(string);
```

string

The string to use for this output state.

Unsupported Procedure

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

```
unsupported(string);
```

string
The string to use for this output state.

Init_testcounts Procedure

```
init_testcounts();
```

Incr_count Procedure

```
incr_count(name args);
```

name

args

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 condition 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 any set of the options matches, then this conditional is true.

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.

Example 6-1. 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 independantly of each other, so a "-Wall -v" matches either "-Wall -v" or "-v -Wall". A space seperates the options in the string. Glob-style regular expressions 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 configuration triplets to clear.

Verbose Procedure

Test cases can use this function to issue helpful messages depending on the number of `--verbose` options on the runtest command line. It prints string if the value of the variable verbose is higher than or equal to the optional number. The default value for number is *1*. Use the optional `-log` argument to cause string to always be added to the log file, even if it won't be printed. Use the optional `-n` argument to print string without a trailing newline. Use the optional `--` argument if string begins with "-".

```
verbose(-log -n -r string number);
```

-log

-n

--

string

number

Load_lib Procedure

Loads a DejaGnu library file by searching a fixed path 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.

Procedures For Remote Communication

`lib/remote.exp` defines these functions, for establishing and managing communications. Each of these procedures tries to establish the connection up to three times before returning. Warnings (if retries will continue) or errors (if the attempt is abandoned) report on communication failures. The result for any of these procedures is either *-1*, when the connection cannot be established, or the spawn ID returned by the Expect command **spawn**.

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

Call_remote Procedure

```
call_remote(type proc dest args);
```

proc

dest

args

Check_for_board_status Procedure

```
check_for_board_status(variable);
```

variable

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

`local_exec(`*`commandline inp outp timeout`*`);`

inp

outp

timeout

Remote_binary Procedure

`remote_binary(`*`host`*`);`

host

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

`remote_download(`*`dest file args`*`);`

dest

file

args

Remote_exec Procedure

`remote_exec(`*`hostname program args`*`);`

hostname

program

args

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

`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

```
remote_pop_conn(host);
```

host

Remote_push_conn Procedure

```
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

`remote_reboot(`*`host`*`);`

`host`

Remote_send Procedure

`remote_send(`*`dest string`*`);`

`dest`

`string`

Remote_spawn Procedure

`remote_spawn(`*`dest commandline args`*`);`

`dest`

`commandline`

`args`

Remote_swap_conn Procedure

`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

`remote_wait(dest timeout);`

dest

timeout

Standard_close Procedure

`standard_close(host);`

host

Standard_download Procedure

`standard_download(`*`dest file destfile`*`);`

dest

file

destfile

Standard_exec Procedure

`standard_exec(`*`hostname args`*`);`

hostname

args

Standard_file Procedure

`standard_file(`*`destopargs`*`);`

Standard_load Procedure

`standard_load(`*`dest prog args`*`);`

dest

prog

args

Standard_reboot Procedure

```
standard_reboot(host);
```

host

Standard_send Procedure

```
standard_send(dest string);
```

dest

string

Standard_spawn Procedure

```
standard_spawn(dest commandline);
```

dest

commndline

Standard_transmit Procedure

```
standard_transmit(dest file);
```

dest

file

Standard_upload Procedure

`standard_upload(`*dest srcfile destfile*`);`

dest

srcfile

destfile

Standard_wait Procedure

`standard_wait(`*dest timeout*`);`

dest

timeout

Unix_clean_filename Procedure

`unix_clean_filename(`*dest file*`);`

dest

file

Procedures For Using Utilities to Connect

telnet, rsh, tip, kermit

telnet Procedure

`telnet(`*`hostname port`*`);`

`rlogin(`*`hostname`*`);`

rsh Procedure

`rsh(`*`hostname`*`);`

`hostname`

This refers to the IP address or name (for example, an entry in `/etc/hosts`) for this target. The procedure names reflect the Unix utility used to establish a connection. The optional *`port`* is used to specify the IP port number. The value of the *`netport`* field in the target_info array is used. (was $netport) This value has two parts, the hostname and the port number, seperated by a :. If host or target is used in the hostname field, than the config array is used for all information.

Tip Procedure

`tip(`*`port`*`);`

`port`

Connect using the Unix utility **tip**. *`Port`*must be a name from the tip configuration file `/etc/remote`. Often, this is called hardwire, or something like ttya. This file holds all the configuration data for the serial port. The value of the serial field in the target_info array is used. (was $serialport) If `host` or `target` is used in the *`port`* field, than the config array is used for all information. the config array is used for all information.

Kermit Procedure

`kermit(`*`port bps`*`);`

`port`

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

`bps`

`bps` is the line speed to use (in its per second) for the connection. The value of the serial field in the target_info array is used. (was $serialport) If `host` or `target`

is used in the *port* field, than the config array is used for all information. the config array is used for all information.

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

`telnet_open(`*hostname args*`);`

hostname

args

Telnet_binary Procedure

`telnet_binary(`*hostname*`);`

hostname

Telnet_transmit Procedure

`telnet_transmit(`*dest file args*`);`

dest

file

args

Tip_open Procedure

```
tip_open(hostname);
```

hostname

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

Ftp_open Procedure

`ftp_open(host);`

host

Ftp_upload Procedure

`ftp_upload(`*`host remotefile localfile`*`);`

host

remotefile

localfile

Ftp_download Procedure

`ftp_download(`*`host localfile remotefile`*`);`

host

localfile

remotefile

Ftp_close Procedure

`ftp_close(`*`host`*`);`

host

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 downlaod.

Procedures For Target Boards

Default_link Procedure

`default_link(board objects destfile flags);`

board

objects

destfile

flags

Default_target_assemble Procedure

`default_target_assemble(source destfile flags);`

source

destfile

flags

default_target_compile Procedure

`default_target_compile(`*`source destfile type options`*`);`

source

destfile

type

options

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_target();
```

Target_assemble Procedure

```
target_assemble(source destfile flags);
```

source

destfile

flags

Target_compile Procedure

```
target_compile(source destfile type options);
```

source

destfile

type

options

Target Database Procedures

Board_info Procedure

`board_info(`*machine op args*`);`

machine

op

args

Host_info Procedure

`host_info(`*op args*`);`

op

args

Set_board_info Procedure

`set_board_info(`*entry value*`);`

entry

value

Set_currtarget_info Procedure

```
set_currtarget_info(entry value);
```

entry

value

Target_info Procedure

```
target_info(op args);
```

op

args

Unset_board_info Procedure

```
unset_board_info(entry);
```

entry

Unset_currtarget_info Procedure

```
unset_currtarget_info(entry);
```

entry

Push_target Procedure

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

```
push_target(name);
```

name

The name of the target to make current connection.

Pop_target Procedure

This unsets the current target connection.

```
pop_target();
```

List_targets Procedure

This lists all the supported targets for this architecture.

```
list_targets();
```

Push_host Procedure

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

```
push_host(name);
```

name

Pop_host Procedure

This unsets the current host connection.

```
pop_host();
```

Compile Procedure

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

```
compile(file);
```

file

Archive Procedure

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

```
archive(file);
```

file

Ranlib Procedure

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

```
ranlib(file);
```

file

Execute_anywhere Procedure

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

```
execute_anywhere(cmdline);
```

cmdline

Platform Dependant Procedures

Each combination of target and tool requires some target-dependent procedures. The names of these procedures have a common form: the tool name, followed by an underbar _, and finally a suffix describing the procedure's purpose. For example, a procedure to extract the version from GDB is called gdb_version.

runtest itself calls only two of these procedures, ${tool}_exit and ${tool}_version; these procedures use no arguments.

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

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();
```

Utility Procedures

Getdirs Procedure

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

```
getdirs(rootdir pattern);
```

args

pattern

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

Find Procedure

Search for files whose names match *pattern* (using shell wildcard characters for filename expansion). Search subdirectories recursively, starting at *rootdir*. The result is the list of files whose names match; if no files match, the result is empty. Filenames in the result include all intervening subdirectory names. If no files match the pattern, then a NULL string is returned.

```
find(rootdir pattern);
```

rootdir

The top level directory to search the search from.

pattern

A csh "glob" style regular expression reprsenting the files to find.

Which Procedure

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

```
which(file);
```

binary

The executable program or shell script to look for.

Grep Procedure

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

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

```
grep(filename regexp --line);
```

filename

The file to search.

regexp

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

--line

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

Prune Procedure

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

```
prune(list pattern);
```

list

A Tcl list containing the original data. Commonly this is the output of a batch executed command, like running a compiler.

pattern

The csh shell "glob" style pattern to search for.

Slay Procedure

This look in the process table for *name* and send it a unix SIGINT, killing the process. This will only work under NT if you have Cygwin or another Unix system for NT installed.

```
slay(name);
```

name
: The name of the program to kill.

Absolute Procedure

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

```
absolute(path);
```

path
: The path to convert.

Psource Procedure

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

```
psource(file);
```

filename
: The filename to Tcl script to source.

Runtest_file_p Procedure

Search *runtest*s for *testcase* and return *1* if found, *0* if not. *runtests* is a list of two elements. The first is a copy of what was on the right side of the = if

```
foo.exp="..."
```

" was specified, or an empty string if no such argument is present. The second is the pathname of the current testcase under consideration. This is used by tools like compilers where each testcase is a file.

```
runtest_file_p(runtests testcase);
```

runtests
: The list of patterns to compare against.

testcase
: The test case filename.

Diff Procedure

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

```
diff(file_1 file_2);
```

file_1
: The first file to compare.

file_2
: The second file to compare.

Setenv Procedure

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

```
setenv(var val);
```

var
: The environment variable to set.

val
: The value to set the variable to.

unsetenv Procedure

Unsets the environment variable *var*.

```
unsetenv(var);
```

var
: The environment variable to unset.

Getenv Procedure

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

```
getenv(var);
```

var
: The environment variable to get the value of.

Prune_system_crud Procedure

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

```
prune_system_crud(system test);
```

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

text
: The Tcl variable containing the text.

Libgloss, A Free BSP

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

Libgloss_link_flags Procedure

```
libgloss_link_flags(args);
```

args

Libgloss_include_flags Procedure

```
libgloss_include_flags(args);
```

args

Newlib_link_flags Procedure

```
newlib_link_flags(args);
```

args

Newlib_include_flags Procedure

```
newlib_include_flags(args);
```

args

Libio_include_flags Procedure

```
libio_include_flags(args);
```

args

Libio_link_flags Procedure

```
libio_link_flags(args);
```

args

G++_include_flags Procedure

```
g++_include_flags(args);
```

args

G++_link_flags Procedure

```
g++_link_flags(args);
```

args

Libstdc++_include_flags Procedure

`libstdc++_include_flags(`*`args`*`);`

args

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

`find_gcc();`

Find_gcj Procedure

```
find_gcj();
```

Find_g++ Procedure

```
find_g++();
```

Find_g77 Procedure

```
find_g77();
```

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

Procedures for debugging your Tcl code.

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

Dumpvars Procedure

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

```
dumpvars(vars);
```

vars

The variables to dump.

Dumplocals Procedure

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

```
dumplocals(args);
```

args

Dumprocs Procedure

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

```
dumprocs(pattern);
```

pattern

The csh "glob" style pattern to look for.

Dumpwatch Procedure

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

```
dumpwatch(pattern);
```

pattern

The csh "glob" style pattern to look for.

Watcharray Procedure

```
watcharray(element type);
```

type

The csh "glob" style pattern to look for.

Watchvar Procedure

```
watchvar(var type);
```

Watchunset Procedure

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

```
watchunset(arg);
```

args

Watchwrite Procedure

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

```
watchwrite(var);
```

var

The variable to watch.

Watchread Procedure

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

```
watchread(var);
```

var

The variable to watch.

Watchdel Procedure

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

```
watchdel(args);
```

args

Print Procedure

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

```
print(var);
```

var

Quit Procedure

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

```
quit();
```

File Map

This is a map of the files in DejaGnu.

- runtest
- runtest.exp
- stub-loader.c
- testglue.c
- config
- baseboards
- lib/debugger.exp
- lib/dg.exp
- lib/framework.exp
- lib/ftp.exp
- lib/kermit.exp
- lib/libgloss.exp
- lib/mondfe.exp
- lib/remote.exp
- lib/rlogin.exp
- lib/rsh.exp
- lib/standard.exp
- lib/target.exp
- lib/targetdb.exp
- lib/telnet.exp

- lib/tip.exp
- lib/util-defs.exp
- lib/utils.exp
- lib/xsh.exp
- lib/dejagnu.exp

Notes

1. http://www.gnu.org
2. http://www.gnu.org/software/dejagnu/

Chapter 7. Unit Testing API

C Unit Testing API

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

Pass Function

This prints a message for a successful test completion.

```
pass(msg);
```

Fail Function

This prints a message for an unsuccessful test completion.

```
fail(msg);
```

Untested Function

This prints a message for an test case that isn't run for some technical reason.

```
untested(msg);
```

Unresolved Function

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

```
unresolved(msg);
```

Totals Function

This prints out the total numbers of all the test state outputs.

```
totals();
```

C++ Unit Testing API

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

Pass Method

This prints a message for a successful test completion.

```
TestState::pass(msg);
```

Fail Method

This prints a message for an unsuccessful test completion.

```
TestState::fail(msg);
```

Untested Method

This prints a message for an test case that isn't run for some technical reason.

```
TestState::untested(msg);
```

Unresolved Method

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

```
TestState::unresolved(msg);
```

Totals Method

This prints out the total numbers of all the test state outputs.

```
TestState::totals();
```