DejaGnu
1 Abstract

This document describes the functionality of DejaGnu, the testing framework of the GNU project. DejaGnu is written in Expect, which uses Tcl as a command language. Expect acts as a very programmable shell. As with other Unix command shells, you can run any program, but once the program is started, your test script has programmable control over its input and output. This does not just apply to the programs under test; expect can also run any auxiliary program, such as diff or sh, with full control over its input and output.

DejaGnu itself is merely a framework for the creation of testsuites. Testsuites are distributed with each application.
2 Overview

2.1 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 testsuites, 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 web site and the Expect web site is at NIST.

Julia Menapace first coined the term “DejaGnu” 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. More importantly, it was also named after my daughter, Deja Snow Savoye (now 14 years old as of Feb 2004), who was a toddler during DejaGnu’s beginnings.

DejaGnu offers several advantages for testing:

- The flexibility and consistency of the DejaGnu framework make it easy to write tests for any program, with either batch oriented, or interactive programs.
- DejaGnu provides a layer of abstraction which allows you to write tests that are portable to any host or target where a program must be tested. For instance, a test for GDB can run from any supported host system on any supported target system. DejaGnu runs tests on many single board computers, whose operating software ranges from a simple boot monitor to a real-time OS.
- All tests have the same output format. This makes it easy to integrate testing into other software development processes. DejaGnu’s output is designed to be parsed by other filtering script and it is also human readable.
- Using Tcl and Expect, it’s easy to create wrappers for existing testsuites. By incorporating existing tests under DejaGnu, it’s easier to have a single set of report analyse programs.

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

2.2 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, it has made that subsystem very complicated. The goal is to have this entirely rewritten using iTcl by the next release. Other changes are:
• More built-in 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 Section 8.3.8 [Libgloss], page 73.
• Lots of little bug fixes from years of heavy use at Cygnus Solutions.
• DejaGnu now uses Automake for Makefile configuration.
• Updated documentation, now in DocBook XML.
• Windows support. There is beta level support for Windows that is still a work in progress. This requires the Cygwin POSIX subsystem for Windows.

2.2.1 Windows Support
To use DejaGnu on Windows, you need to first install the Cygwin release. This works as of the B20.1 release. Cygwin is a POSIX system for Windows. This covers both utility programs and a library that adds POSIX system calls to Windows. Among them is pseudotty support for Windows that emulates the POSIX pty standard. The latest Cygwin is always available from this location. This works well enough to run "make check" of the GNU development tree on Windows after a native build. But the nature of ptys on Windows is still evolving. Your mileage may vary.

2.3 Design Goals
DejaGnu grew out of the internal needs of Cygnus Solutions, the company formerly known as 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;
• permitted testing interactive programs, like GDB; and
• permitted 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. 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.

2.4 A POSIX conforming test framework
DejaGnu conforms to the POSIX 1003.3 standard for test frameworks. Rob Savoye was a member of that committee.
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 day or night. It is important to note that the standard being tested is never 1003.3; the standard being tested is some other standard, for which the assertions were written.

As there is no testsuite to 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 testsuite 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 commands like return—if you alter the flow of control of the Tcl code you must insure that every test still produces some result code.

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

- 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 place-holder, 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 testsuites and these procedures are already known to conform to POSIX 1003.3. For a DejaGnu testsuite to conform to POSIX 1003.3, you must avoid the `setupxfail` 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.
3 Getting DejaGnu up and running

This chapter was originally written by Niklaus Giger (ngiger@mus.ch) because he lost a week to figure out how DejaGnu works and how to write a first test.

Follow these instructions as closely as possible in order get a good insight into how DejaGnu works, else you might run into a lot of subtle problems. You have been warned.

It should be no big problems installing DejaGnu using your package manager or from the source code. On the Debian GNU/Linux system just run (as root):

    apt-get install dejagnu

These examples were run on a primary machine with a AMD K6 and a Mac PowerBook G3 serving as a remote target.

The tests for Windows were run under Windows using the actual Cygwin version (1.3.x as of October 2001). Its target system was a PPC embedded system running vxWorks.

3.1 Test your installation

Create a new user called "dgt" (DejaGnuTest), which uses bash as it login shell. PS1 must be set to "\u: \w\$ " in its ~/.bashrc. Login as this user, create an empty directory and change the working directory to it. e.g

    dgt:~$ mkdir ~/dejagnu.test
    dgt:~$ cd ~/dejagnu.test

Now you are ready to test DejaGnu's main program called runtest. The expected output is shown

    Runtest output in a empty directory

    dgt:~/dejagnu.test$ runtest
    WARNING: Couldn't find the global config file.
    WARNING: No tool specified Test
    Run By dgt on Sun Nov 25 17:07:03 2001 Native configuration is i586-pc-linux-gnu
    === tests ===
    Schedule of variations: unix
    Running target unix Using /usr/share/dejagnu/baseboards/unix.exp as board description file for target.
    Using /usr/share/dejagnu/config/unix.exp as generic interface file for target.
    ERROR: Couldn't find tool config file for unix.
    === Summary ===

    We will show you later how to get rid of all the WARNING- and ERROR-messages. The files testrun.sum and testrun.log have been created, which do not interest us at this point. Let's remove them.

    dgt:~/dejagnu.test$ rm testrun.sum testrun.log

3.1.1 Windows

On Windows systems DejaGnu is part of a port of a lot of Unix tools to the Windows OS, called Cygwin. Cygwin may be downloaded and installed from a mirror of
http://www.cygwin.com/. All examples were also run on Windows. If nothing is said, you can assume that you should get the same output as on a Unix system.

You will need a telnet daemon if you want to use a Windows box as a remote target. There seems to be a freeware telnet daemon at http://www.fictional.net/.

3.1.2 Getting the source code for the calc example

If you are running a Debian distribution you can find the examples under /usr/share/doc/dejagnu/examples. These examples seem to be missing in Red Hat’s RPM. In this case download the sources of DejaGnu and adjust the paths to the DejaGnu examples accordingly.

3.2 Create a minimal project, e.g. calc

In this section you will to start a small project, using the sample application calc, which is part of your DejaGnu distribution

3.2.1 A simple project without the GNU autotools

The runtest program can be run stand-alone. All the autoconf/automake support is just cause those programs are commonly used for other GNU applications. The key to running runtest stand-alone is having the local site.exp file setup correctly, which automake does.

The generated site.exp should like like:

```bash
set tool calc
set srcdir .
set objdir /home/dgt/dejagnu.test
```

3.2.2 Using autoconf/autoheader/automake

We have to prepare some input file in order to run autoconf and automake. There is book "GNU autoconf, automake and libtool" by Garry V. Vaughan, et al. NewRider, ISBN 1-57870-190-2 which describes this process thoroughly.

From the calc example distributed with the DejaGnu documentation you should copy the program file itself (calc.c) and some additional files, which you might examine a little bit close to derive their meanings.

```bash
dgt:~/dejagnu.test$ cp -r /usr/share/doc/dejagnu/examples/calc/\{configure.in,Makefile.am,calc.c,testsuite\} .
```

In Makefile.am note the presence of the AUTOMAKE_OPTIONS = dejagnu. This option is needed.

Run aclocal to generate aclocal.m4, which is a collection of macros needed by configure.in

```bash
dgt:~/dejagnu.test$ aclocal
```

autoconf is another part of the auto-tools. Run it to generate configure based on information contained in configure.in.

```bash
dgt:~/dejagnu.test$ autoconf
```
autoheader is another part of the auto-tools. Run it to generate calc.h.in.

    dgt:~/$ autoheader

The Makefile.am of this example was developed as part of the DejaGnu distribution. Adapt Makefile.am for this test. Replace the line "#noinst_PROGRAMS = calc" to "bin_PROGRAMS = calc". Change the RUNTESTDEFAULTFLAGS from "$srcdir/testsuite" to "/testsuite".

    dgt:~/$ autoheader

Running automake at this point contains a series of warning in its output as shown in the following example:

**Sample output of automake with missing files**

    dgt:~/$ automake --add-missing
    automake: configure.in: installing './install-sh'
    automake: configure.in: installing './mkinstalldirs'
    automake: configure.in: installing './missing'
    automake: Makefile.am: installing './INSTALL'
    automake: Makefile.am: required file './NEWS' not found
    automake: Makefile.am: required file './README' not found
    automake: Makefile.am: installing './COPYING'
    automake: Makefile.am: required file './AUTHORS' not found
    automake: Makefile.am: required file './ChangeLog' not found
    configure.in: 4: required file './calc.h.in' not found
    Makefile.am:6: required directory ./doc does not exist

    dgt:~/$ automake --add-missing
    automake: configure.in: installing './install-sh'
    automake: configure.in: installing './mkinstalldirs'
    automake: configure.in: installing './missing'
    automake: Makefile.am: installing './INSTALL'
    automake: Makefile.am: required file './NEWS' not found
    automake: Makefile.am: required file './README' not found
    automake: Makefile.am: installing './COPYING'
    automake: Makefile.am: required file './AUTHORS' not found
    automake: Makefile.am: required file './ChangeLog' not found
    configure.in: 4: required file './calc.h.in' not found
    Makefile.am:6: required directory ./doc does not exist

Create a empty directory doc and empty files INSTALL, NEWS, README, AUTHORS, ChangeLog and COPYING. The default COPYING will point to the GNU Public License (GPL). In a real project it would be time to add some meaningful text in each file.

Adapt calc to your environment by calling configure.

**Sample output of configure**

    dgt:~/$ ./configure
    creating cache ./config.cache
    checking whether to enable maintainer-specific portions of Makefiles... no
    checking for a BSD compatible install... /usr/bin/install -c
    checking whether build environment is sane... yes
    checking whether make sets ${MAKE}... yes
    checking for working aclocal... found
    checking for working autoconf... found
    checking for working automake... found
    checking for working autoheader... found
    checking for working autoheader... found
    checking for working autotest... found
    checking for gfortran... found
    checking whether the C compiler (gcc ) works... yes
    checking whether the C compiler (gcc ) is a cross-compiler... no
    checking whether we are using GNU C... yes
    checking whether gcc accepts -g... yes
    checking for a BSD compatible install... /usr/bin/install -c
checking how to run the C preprocessor... gcc -E
checking for stdlib.h... yes
checking for strcmp... yes
updating cache ./config.cache
creating ./config.status
creating Makefile creating calc.h

If you are familiar with GNU software, this output should not contain any surprise to you. Any errors should be easy to fix for such a simple program.

Build the calc executable:

**Sample output building calc**

```bash
dgt:~/dejagnu.test$ make
  gcc -DHAVE_CONFIG_H -I. -I. -I. -g -O2 -c calc.c
  gcc -g -O2 -o calc calc.o
```

You prepared a few files and then called some commands. Respecting the right order assures a automatic and correctly compiled calc program. The following example resumes the correct order.

**Creating the calc program using the GNU autotools**

```bash
dgt:~/dejagnu.test$ aclocal
dgt:~/dejagnu.test$ autoconf
dgt:~/dejagnu.test$ autoheader
dgt:~/dejagnu.test$ automake --add-missing
dgt:~/dejagnu.test$ ./configure
dgt:~/dejagnu.test$ make
```

Play with calc and verify whether it works correctly. A sample session might look like this:

```bash
dgt:~/dejagnu.test$ ./calc
  calc: version
  Version: 1.1
  calc: add 3 4
    7
calc: multiply 3 4
    12
calc: multiply 2 4
    12
calc: quit
```

Look at the intentional bug that 2 times 4 equals 12.

The tests run by DejaGnu need a file called site.exp, which is automatically generated if we call "make site.exp". This was the purpose of the "AUTOMAKE_OPTIONS = dejagnu" in Makefile.am.

**Sample output generating a site.exp**
3.3 Our first automated tests

3.3.1 Running the test for the calc example

Now we are ready to call the automated tests

Sample output of runtest in a configured directory

```
dgt:~/dejagnu.test$ make check
make check-DEJAGNU
make[1]: Entering directory '/home/dgt/dejagnu.test' srcdir='cd . & & pwd'; export srcdir
EXPECT=expect; export EXPECT; \ runtest=runtest; \ 
if /bin/sh -c "$runtest --version" > /dev/null 2>&1; then \ 
$runtest --tool calc CALC='pwd'/calc --srcdir ./testsuite ; \ 
else echo "WARNING: could not find \'runtest\'" 1>&2; ;;\ 
fi
WARNING: Couldn't find the global config file.
WARNING: Couldn't find tool init file
Test Run By dgt on Sun Nov 25 21:42:21 2001
Native configuration is i586-pc-linux-gnu

=== calc tests ===

Schedule of variations:

unix

Running target unix
Using /usr/share/dejagnu/baseboards/unix.exp as board description file for target.
Using /usr/share/dejagnu/config/unix.exp as generic interface file for target.
Using ./testsuite/config/unix.exp as tool-and-target-specific interface file.
Running ./testsuite/calc.test/calc.exp ... 
FAIL: multiply2 (bad match)

=== calc Summary ===

# of expected passes 5
# of unexpected failures 1
/home/Dgt/dejagnu.test/calc version Version: 1.1
make[1]: *** [check-DEJAGNU] Fehler 1
make[1]: Leaving directory '/home/Dgt/dejagnu.test' make: *** [check-am] Fehler 2
```

Did you see the line "FAIL:"? The test cases for calc catch the bug in the calc.c file. Fix the error in calc.c later as the following examples assume a unchanged calc.c.
Examine the output files calc.sum and calc.log. Try to understand the test cases written in ~/dejagnu.test/testsuite/calc.test/calc.exp. To understand Expect you might take a look at the book "Exploring Expect", which is an excellent resource for learning and using Expect. (Pub: O’Reilly, ISBN 1-56592-090-2) The book contains hundreds of examples and also includes a tutorial on Tcl. Exploring Expect is 602 pages long.

3.3.2 The various config files or how to avoid warnings

DejaGnu may be customized by each user. It first searches for a file called ~/.dejagnurc. Create the file ~/.dejagnurc and insert the following line:

```bash
puts "I am ~/.dejagnurc"
```

Rerun make check. Test whether the output contains "I am ~/.dejagnurc". Create ~/my_dejagnu.exp and insert the following line:

```bash
puts "I am ~/my_dejagnu.exp"
```

In a Bash-Shell enter

```bash
dgt:~/dejagnu.test$ export DEJAGNU=~/my_dejagnu.exp
```

Run "make check" again. The output should not contain "WARNING: Couldn’t find the global config file.". Create the sub-directory lib. Create the file "calc.exp" in it and insert the following line:

```bash
puts "I am lib/calc.exp"
```

The last warning "WARNING: Couldn’t find tool init file" should not be part of the output of make check. Create the directory ~/boards. Create the file ~/boards/standard.exp and insert the following line:

```bash
puts "I am boards/standard.exp"
```

If the variable DEJAGNU is still not empty then the (abbreviated) output of "make check" should look like this:

Sample output of runtest with the usual configuration files

```bash
dgt:~/dejagnu.test$ make check
<...>
fi
I am ~/.dejagnurc
I am ~/my_dejagnu.exp
I am lib/calc.exp
Test Run By dgt on Sun Nov 25 22:19:14 2001
Native configuration is i586-pc-linux-gnu

== calc tests ==
Using /home/Dgt/boards/standard.exp as standard board description file for build.
```
I am ~/boards/standard.exp
Using /home/Dgt/boards/standard.exp as standard board description
file for host.
I am ~/boards/standard.exp

Schedule of variations:
  unix

Running target unix
Using /home/Dgt/boards/standard.exp as standard board description
file for target.
I am ~/boards/standard.exp
Using /usr/share/dejagnu/baseboards/unix.exp as board description file
for target.
<...>

It is up to you to decide when and where to use any of the above mentioned config files for customizing. This chapters showed you where and in which order the different config files are run.

3.3.3 When trouble strikes
Calling runtest with the '-v'-flag shows you in even more details which files are searched in which order. Passing it several times gives more and more details.

Displaying details about runtest execution

    runtest -v -v --tool calc CALC='pwd'/calc --srcdir ./testsuite

Calling runtest with the '-debug'-flag logs a lot of details to dbg.log where you can analyse it afterwards.

In all test cases you can temporary adjust the verbosity of information by adding the following Tcl command to any Tcl file that gets loaded by dejagnu, for instance, ~/.dejagnurc:

    set verbose 9

3.3.4 Testing "Hello world" locally
This test checks whether the shell command echo Hello world will really output "Hello world" to the console. Create the file '~/dejagnu.test/testsuite/calc.test/local_echo.exp'. It should contain the following lines:

A first (local) test case

    set test "Local Hello World"
    spawn echo Hello World
    expect {
        "Hello World" { pass $test }
        default { fail $test }
    }
Run runtest again and verify the output "calc.log"

3.4 A first remote test

Testing remote targets is a lot trickier especially if you are using an embedded target which has no built in support for things like a compiler, FTP server or a Bash-shell. Before you can test calc on a remote target you have to acquire a few basics skills.

3.4.1 Setup telnet to your own host

The easiest remote host is usually the host you are working on. In this example we will use telnet to login in your own workstation. For security reasons you should never have a telnet daemon running on machine connected on the Internet, as password and user names are transmitted in clear text. We assume you know how to setup your machine for a telnet daemon.

Next try whether you may login in your own host by issuing the command "telnet localhost.1". In order to be able to distinguish between a normal session and a telnet login add the following lines to /home/dgt/.bashrc.

```bash
if [ "$REMOTEHOST" ]
then
    PS1='remote:\w\$ '
fi
```

Now on the machine a "remote" login looks like this:

**Sample log of a telnet login to localhost**

```
dgt:~$dejagnu.test$ telnet localhost
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is "^]."
Debian GNU/Linux testing/unstable Linux
K6Linux login: dgt
Password:
Last login: Sun Nov 25 22:46:34 2001 from localhost on pts/4
Linux K6Linux 2.4.14 #1 Fre Nov 16 19:28:25 CET 2001 i586 unknown
No mail.
remote:"$ exit
logout
Connection closed by foreign host.
```

3.4.2 A test case for login via telnet

In order to define a correct setup we have add a line containing "set target unix" either to "/dejagnurc" or to "/my_dejagnu.exp. In "/boards/standard.exp" add the following four lines to define a few patterns for the DejaGnu telnet login procedure.

**Defining a remote target board**
set_board_info shell_prompt  "remote:"
set_board_info telnet_username "dgt"
set_board_info telnet_password "top_secret"
set_board_info hostname "localhost"

As DejaGnu will be parsing the telnet session output for some well known pattern the output there are a lot of things that can go wrong. If you have any problems verify your setup:

• Is ‘/etc/motd’ empty?
• Is ‘/etc/issue.net’ empty?
• Exists a empty ‘~/.hushlogin’?
• The LANG environment variable must be either empty or set to "C".

To test the login via telnet write a sample test case. Create the file “~/dejagnu.test/testsuite/calc.test/remote_echo.exp” and add the following few lines:

DejaGnu script for logging in into a remote target

puts "this is remote_echo.exp target for $target "
target_info $target
#set verbose 9
set shell_id [remote_open $target]
set test "Remote login to $target"
#set verbose 0
puts "Spawn id for remote shell is $shell_id"
if { $shell_id > 0 } {
    pass "$test"
} else {
    fail "Remote open to $target"
}

In the runtest output you should find something like:

    Running ./testsuite/calc.test/local_echo.exp ...
    Running ./testsuite/calc.test/remote_echo.exp ...
    this is remote_echo.exp target is unix
    Spawn id for remote shell is exp?

Have again a look at calc.log to get a feeling how DejaGnu and expect parse the input.

3.4.3 Remote testing "Hello world"

Next you will transform the above "hello world" example to its remote equivalent. This can be done by adding the following lines to our file remote_echo.exp.

A first (local) remote "Hello world" test

    set test "Remote_send Hello World"
    set status [remote_send $target "echo \"Hello\" \"World\"\n"
    pass "$test"
Call make check. The output should contain "# of expected passes 9" and "# of unexpected failures 1".

Have a look at the procedures in /usr/share/dejagnu/remote.exp to have an overview of the offered procedures and their features.

Now set up a real target. In the following example we assume as target a PowerBook running Debian. As above add a test user "dgt", install Telnet and FTP servers. In order to distinguish it from the host add the line

```
PS1='test:>'
```

to /home/dgt/.bash_profile. Also add a corresponding entry "powerbook" to /etc/hosts and verify that you are able to ping, telnet and ftp to the target "powerbook".

In order to let runtest run its test on the "powerbook" target change the following lines in ~/boards/standard.exp:

```
Board definition for a remote target

set_board_info protocol      "telnet"
set_board_info telnet_username "dgt"
set_board_info telnet_password "top_secret"
set_board_info shell_prompt   "test:> "
set_board_info hostname       "powerbook"
```

Now call runtest again with the same arguments and verify whether all went okay by taking a close look at calc.log.

### 3.4.4 Transferring files from/to the target

A simple procedure like this will do the job for you:

**Test script to transfer a file to a remote target**

```
set test "Remote_download"
puts "Running Remote_download"
# set verbose 9
set remfile /home/dgt/dejagnu2

set status [remote_download $target /home/dgt/.dejagnurc $remfile]
if { "$status" == "" } {
    fail "Remote download to $remfile on $target"
} else {
    pass "$test"
}

puts "status of remote_download ist $status"
# set verbose 0
```
After running runtest again, check whether the file dejagnu2 exists on the target. This example will only work if the rcp command works with your target. If you have a working FTP-server on the target you can use it by adding the following lines to `~/boards/standard.exp:

### Defining a board to use FTP as file transport

```bash
set_board_info file_transfer "ftp"
set_board_info ftp_username "dgt"
set_board_info ftp_password "1234"
```

#### 3.4.5 Preparing for crosscompilation

For crosscompiling you need working binutils, gcc and a base library like libc or glib for your target. It is beyond the scope of this document to describe how to get it working. The following examples assume a cross compiler for PowerPC which is called linux-powerpc-gcc.

Add `AC_CANONICAL_TARGET` in `dejagnu.test/configure.in` at the following location. Copy `config.guess` from `/usr/share/automake` to `dejagnu.test`.

```bash
AM_CONFIG_HEADER(calc.h)
AC_CANONICAL_TARGET([])
AM_INIT_AUTOMAKE(calc, 1.1)
```

You need to run automake 2.5 or later. Depending on your installation calling `autoconf2.5` instead of `autoconf` is not needed. The sequence to regenerate all files is:

### Using autotools for cross development

```bash
$ autoconf2.5
$ autoheader
$ automake
$ ./configure --host=powerpc-linux --target=powerpc-linux
configure: WARNING: If you wanted to set the --build type, don’t use --host.
If a cross compiler is detected then cross compile mode will be used.
checking build system type... ./config.guess: ./config.guess: No such file or directory;
configure: error: cannot guess build type; you must specify one
$ cp /usr/share/automake/config.guess .
$ ./configure --host=powerpc-linux --target=powerpc-linux
configure: WARNING: If you wanted to set the --build type, don’t use --host.
If a cross compiler is detected then cross compile mode will be used.
checking build system type... i586-pc-linux-gnu
checking host system type... powerpc-unknown-linux-gnu
<...>
checking whether we are cross compiling... yes
<...>
Configuration:
Source code location: .
C Compiler: powerpc-linux-gcc
C Compiler flags: -g -O2
```
Everything should be ready to recompile for the target:

$ make
  powerpc-linux-gcc -DHAVE_CONFIG_H -I. -I. -I. -g -O2 -c calc.c
  powerpc-linux-gcc -g -O2 -o calc calc.o

3.4.6 Remote testing of calc
Not yet written, as I have problem getting libc6-dev-powerpc to work. Probably I first have
to build my cross compiler.

3.4.7 Using Windows as host and vxWorks as target
A more thorough walk-through will be written in a few weeks.

  In order to test the vxWorks as a target I changed boards/standards.exp to reflect my
  settings (IP, username, password). Then I reconfigured vxWorks to include a FTP and
telnet server (using the same username/password combination ad in boards/standard.exp).

  With this setup and some minor modification (e.g. replacing echo by printf) in my test
cases I could test my vxWorks system. It sure does not seem to be a correct setup by
DejaGnu standard. For instance, it still loading /usr/share/dejagnu/baseboards/unix.exp
instead of vxWorks. In any case I found that (at least under Windows) I did not find out
how the command line would let me override settings in my personal config files.
4 Running Tests

There are two ways to execute a testsuite. 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` directly from the command line requires either all the correct options, or the Section 5.1 [Local Config File], page 28 must be setup correctly.

4.1 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 determine 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`.

4.2 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.

4.2.1 Output States

`runtest` flags the outcome of each test as one of these cases. Section 2.4 [A POSIX Conforming Test Framework], page 4 for a discussion of how POSIX specifies the meanings of these cases.

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

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

**FAIL** A test failed, although it was expected to succeed. This may indicate regress; inspect the test case and the failing software to locate the bug.
XFAIL  A test failed, but it was expected to fail. This result indicates no change in a known bug. If a test fails because the operating system where the test runs lacks some facility required by the test, the outcome is UNSUPPORTED instead.

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

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

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

runtest may also display the following messages:

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

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

NOTE  An informational message about the test case.

4.2.2 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 separate.

--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 setupxfail are affected by --host. In this usage, host refers to the machine that the tests are to be run on, which may not be the same as the build machine. If --build is also specified, then --host refers to the machine that the tests will be run on, not the machine DejaGnu is run on.

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

--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 testsuite to run, and what initialization module to use. --tool is used only for these two purposes. It is not used to name the executable program to test. Executable tool names (and paths) are recorded in ‘site.exp’ and you can override them by specifying Tcl variables on the command line.

For example, including "--tool gcc" on the 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 Cc).

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.

4.2.3 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 testsuite in the same directory. The default options are in the local site.exp file, created by "make site.exp".

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

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

Test output follows, ending with:

```plaintext
=== 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’.)

4.3 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.

4.3.1 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.

Here is a short sample summary log

    Test Run By rob on Mon May 25 21:40:57 PDT 1992
    === gdb tests ===
    Running ./gdb.t00/echo.exp ...  
    PASS:   Echo test 
    Running ./gdb.all/help.exp ... 

PASS: help add-symbol-file
PASS: help aliases
PASS: help breakpoint "bre" abbreviation
FAIL: help run "r" abbreviation
Running ./gdb.t10/crossload.exp ... 
PASS: m68k-elf (elf-big) explicit format; loaded
XFAIL: mips-ecoff (ecoff-bigmips) "ptype v_signed_char" signed C types

=== gdb Summary ===
# of expected passes 5
# of expected failures 1
# of unexpected failures 1
/usr/latest/bin/gdb version 4.6.5 -q

4.3.2 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.

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
4.3.3 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.

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.

Here is an excerpt from the debugging log for a GDB test:

send: sent {break gdbme.c:34
} 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
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.
5 Customizing DejaGnu

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

DejaGnu supports two ‘site.exp’ files. The multiple instances of ‘site.exp’ are loaded in a fixed order 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 referred to as the “global” config file.

Any directory containing a configured testsuite 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 testsuite 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.

5.1 Local Config File

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

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

The first section starts with

    ## these variables are automatically generated by make ##
    # Do not edit here. If you wish to override these values
    # add them to the last section
In the second section, you can override any default values (locally to DejaGnu) for all the variables. The second section can also contain your preferred defaults for all the command line options to runtest. This allows you to easily customize runtest for your preferences in each configured test-suite tree, so that you need not type options repeatedly on the command line. (The second section may also be empty, if you do not wish to override any defaults.)

The first section ends with this line

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

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

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

Local Config File

## these variables are automatically generated by make ##

# Do not edit here. If you wish to override these values
# add them to the last section
set rootme "/build/devo-builds/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 "-isystem /build/devo-builds/i586-pc-linux-gnulibc1/gcc/../libio -isystem $srcdir/../libg++/src -isystem ...
append LDFLAGS " -L/build/devo-builds/i586-pc-linux-gnulibc1/gcc/../ld"
set tmpdir /build/devo-builds/i586-pc-linux-gnulibc1/gcc/testsuite
set srcdir "${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 exclusively by the GCC testsuite. For most test cases, the CXXFLAGS and LDFLAGS are supplied by DejaGnu itself for cross testing, but to test a compiler, GCC needs to manipulate these itself.

5.2 Global Config File

The master config file is where all the target specific config variables 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 dependent. Here we have two sets of values that we use for testing m68k-aout cross compilers. As both of these target boards has a different debugging protocol, we test on both of them in sequence.

Global Config file

```bash
# 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 Telnet server.

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 Section 6.4 [Adding A New Board], page 40 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.

5.3 Board Config File

The board config file is where board specific 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 Section 6.4 [Adding A New Board], page 40 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. These 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 libraries, or the tool chain itself is part of your build tree.

Board Config File

```plaintext
# 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
```

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 Section 8.3.8 [Libgloss], page 73, 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.

5.4 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/Windows 3.1, MacOS and Windows.

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

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

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

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

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

Remote host setup

```bash
    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 (iusrfoobar), password, and ftp directory (c:/piggy) to match what you selected.

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

**Add The Board Directory**

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

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

**Setup Cross Remote Testing**

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

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

**Setup Native Remote Testing**

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

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

**Run Test Remotely**

```
make -k -w check RUNTESTFLAGS="--host_board foobar --target_board foobar -v -v" > check.out 2>&1
```

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

**Run a Test Remotely**

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

## 5.5 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.

### 5.5.1 Command Line Option Variables

In the user editable second section of the Section 5.5.2 [Personal Config File], page 35 you can not only override the configuration variables captured in the first section, but also specify default values for all on the `runtest` command line options. Save for `--debug`, `--help`, and `--version`, each command line option has an associated Tcl variable. Use the Tcl `set` command to specify a new default value (as for the configuration variables). The following table describes the correspondence between command line options and variables you can set in `site.exp`. Section 4.2.2 [Invoking Runtest], page 20, for explanations of the command-line options.

<table>
<thead>
<tr>
<th>runtest option</th>
<th>Tcl variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–all</td>
<td>all_flag</td>
<td>display all test results if set</td>
</tr>
<tr>
<td>–baud</td>
<td>baud</td>
<td>set the default baud rate to something other than 9600.</td>
</tr>
<tr>
<td>–connect</td>
<td>connectmode</td>
<td><code>rlogin</code>, <code>telnet</code>, <code>rsh</code>, <code>kermit</code>, <code>tip</code>, or <code>mondfe</code></td>
</tr>
<tr>
<td>–outdir</td>
<td>outdir</td>
<td>directory for <code>tool.sum</code> and <code>tool.log</code></td>
</tr>
<tr>
<td>–objdir</td>
<td>objdir</td>
<td>directory for pre-compiled binaries</td>
</tr>
<tr>
<td>–reboot</td>
<td>reboot</td>
<td>reboot the target if set to &quot;1&quot;; do not reboot if set to &quot;0&quot; (the default).</td>
</tr>
<tr>
<td>–srcdir</td>
<td>srcdir</td>
<td>directory of test subdirectories</td>
</tr>
<tr>
<td>–strace</td>
<td>tracelevel</td>
<td>a number: Tcl trace depth</td>
</tr>
<tr>
<td>–tool</td>
<td>tool</td>
<td>name of tool to test; identifies init, test subdir</td>
</tr>
<tr>
<td>–verbose</td>
<td>verbose</td>
<td>verbosity level. As option, use multiple times; as variable, set a number, 0 or greater.</td>
</tr>
<tr>
<td>–target</td>
<td>target_triplet</td>
<td>The canonical configuration string for the target.</td>
</tr>
</tbody>
</table>
-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.

5.5.2 Personal Config File

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

    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.
6 Extending DejaGnu

6.1 Adding A New Testsuite

The testsuite for a new tool should always be located in that tool's 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."

6.2 Adding A New Tool

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

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

There is one testsuite 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 testsuite for it. Reading this testsuite, and experimenting with it, is a good way to supplement the information in this section. (Thanks to Robert Lupton for creating calc and its testsuite—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 testsuite for a program example.

- Create or select a directory to contain your new collection of tests. Change into that directory (shown here as ‘testsuite’):
  Create a ‘configure.in’ file in this directory, to control configuration-dependent choices for your tests. So far as DejaGnu is concerned, the important thing is to set a value for the variable target_abbrev; this value is the link to the init file you will write soon. (For simplicity, we assume the environment is Unix, and use unix as the value.)
  What else is needed in ‘configure.in’ depends on the requirements of your tool, your intended test environments, and which configure system you use. This example is a minimal configure.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 Section 4.1 [Make Check], page 19 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 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.

**Sample Makefile.in Fragment**

```makefile
# 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 \n    echo runtest; \n  fi'

# The flags to pass to runtest
RUNTESTFLAGS =

# Execute the tests
check: site.exp all
  $(RUNTEST) $(RUNTESTFLAGS) \n    --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\n  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
```
• 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:

**Simple Batch Program Tool Init File**

```bash
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:

**Simple Interactive Program Tool Init File**

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

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

Testing A New Tool Config

```tcl
send_user "Testing: one, two...
```

• 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.
• You are now ready to triumphantly type make check or runtest. You should see something like this:

Example Test Case Run

```plaintext
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.

6.3 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.

6.4 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 specific settings go. Commonly there are similar target environments with just different processors.

**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 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

6.5 Board Config File Values

These fields are all in the board_info array. These are all set by using the set_board_info
and add_board_info procedures as required. The parameters are the field name, followed
by the value that the field is set to or is added to the field, respectively.

<table>
<thead>
<tr>
<th>Field</th>
<th>Sample Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler</td>
<td>&quot;[find_gcc]&quot;</td>
<td>The path to the compiler to use.</td>
</tr>
<tr>
<td>cflags</td>
<td>&quot;-mca&quot;</td>
<td>Compilation flags for the compiler.</td>
</tr>
</tbody>
</table>
| ldflags  | "[libgloss_link_flags]
[newlib_link_flags]" | Linking flags for the compiler.                |
| ldscript | "-Wl,-Tidt.ld"                | The linker script to use when cross compiling.  |
| libs     | "-lgcc"                       | Any additional libraries to link in.            |
shell_prompt: "cygmon>
Hex_startaddr: "0xa0020000"
Start_addr: 0xa0008000
StartAddr: "a0020000"
Exit_statuses_bad: 1
Reboot_delay: 10
Unreliable: 1
Sim: [find_sim]
Objcopy: $tempfil
Support_libs: "$\{prefix\_dir\}/i386-coff/
Addl_link_flags: "$-N"

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

**Board Info Fields For GCC & GDB**

<table>
<thead>
<tr>
<th>Field</th>
<th>Sample Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strip</td>
<td>$tempfile</td>
<td>Strip the executable of symbols.</td>
</tr>
<tr>
<td>gdb_load_offset</td>
<td>&quot;0x40050000&quot;</td>
<td>The GDB debugging protocol to use.</td>
</tr>
<tr>
<td>gdb_protocol</td>
<td>&quot;remote&quot;</td>
<td></td>
</tr>
<tr>
<td>gdb_sect_offset</td>
<td>&quot;0x41000000&quot;;</td>
<td>The linker script to use with a GDB stub.</td>
</tr>
<tr>
<td>gdb_stub_ldscript</td>
<td>&quot;-Wl,-Teva-stub.ld&quot;</td>
<td></td>
</tr>
<tr>
<td>gdb_cannot_call_functions</td>
<td>1</td>
<td>Whether GDB can call functions on the target,</td>
</tr>
<tr>
<td>gdb_noargs</td>
<td>1</td>
<td>Whether the target can take command line arguments.</td>
</tr>
<tr>
<td>gdb_nosignals</td>
<td>1</td>
<td>Whether there are signals on the target.</td>
</tr>
<tr>
<td>gdbShortly</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>gdb_start_symbol</td>
<td>&quot;_start&quot;;</td>
<td>The starting symbol in the executable.</td>
</tr>
</tbody>
</table>
6.6 Writing A Test Case

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

6.7 Debugging A Test Case

These are the kinds of debugging information available from DejaGnu:

• Output controlled by test scripts themselves, explicitly allowed for by the test author. This kind of debugging output appears in the detailed output recorded in the DejaGnu log file. To do the same for new tests, use the `verbose' procedure (which in turn uses the 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 \textit{verbose} is 3 or greater, DejaGnu turns on the expect command \texttt{log\_user}. This command prints all expect actions to the expect standard output, to the detailed log file, and (if \texttt{--debug} is on) to 'dbg.log'.

6.8 Adding A Test Case To A Testsuite.

There are two slightly different ways to add a test case. One is to add the test case to an existing directory. The other is to create a new directory to hold your test. The existing test directories represent several styles of testing, all of which are slightly different; examine the 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:

\texttt{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 \texttt{.exp} suffix). Creating a new \texttt{.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 \texttt{configure} and \texttt{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 \texttt{configdirs} definition in the ‘configure.in’ file for the testsuite directory. This way when \texttt{make} and \texttt{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’.

6.9 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 \texttt{expect} command. In this situation, the precise boundary that determines which \texttt{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 \texttt{-re} option for the \texttt{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 \texttt{.*} after the text of interest; this will also match any intervening blank lines. Sometimes an alternative is to match end of line using \texttt{\r} or \texttt{\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.

6.10 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.
7 Unit Testing

7.1 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 application. Also if there is a specification for the API to be tested, the testcase can also function as a compliance test.

7.2 The dejagnu.h Header File

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

8.1 Obtaining DejaGnu

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

8.2 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.

8.2.1 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 separate tree for objects, execute the configure script from the source tree like this:

```
../dejagnu-1.5.2/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 `execprefix`.)

Save for a small number of example tests, the DejaGnu distribution itself does not include any testsuites; these are available separately. Testsuites 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.

8.2.2 Installing DejaGnu

To install DejaGnu in your file system (either in ‘/usr/local’, or as specified by your --prefix option to configure), execute.

```
eg$ make install
```

`make install` does these 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.

8.3 Builtin Procedures
DejaGnu provides these Tcl procedures.

8.3.1 Core Internal Procedures

8.3.1.1 Mail_file Procedure

mail_file(file to subject);

8.3.1.2 Open_logs Procedure

open_logs();

8.3.1.3 Close_logs Procedure

close_logs();

8.3.1.4 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

8.3.1.5 Is_remote Procedure

is_remote(board);

8.3.1.6 is3way Procedure

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

is3way();
8.3.1.7 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);

8.3.1.8 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);

8.3.1.9 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();

8.3.1.10 Unknown Procedure
unknown(args);

args

8.3.1.11 Clone_output Procedure
clone_output(message);

message

8.3.1.12 Reset_vars Procedure
reset_vars();

8.3.1.13 Log_and_exit Procedure
log_and_exit();

8.3.1.14 Log_summary Procedure
log_summary(args);

args

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

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

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

```c
setup_xfail(config bugid);
```

- **config** The config triplet to trigger whether this is an unexpected or expect failure.
- **bugid** The optional bugid, used to tie this test case to a bug tracking system.

8.3.1.17 Record_test Procedure
```c
record_test(type message args);
```

- **type**
- **message**
- **args**

8.3.1.18 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.

```c
pass(string);
```

- **string** The string to use for this PASS message.

8.3.1.19 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.

```c
fail(string);
```

- **string** The string to use for this FAIL message.

8.3.1.20 Xpass Procedure
Declares a test to have unexpectedly 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.

```c
xpass(string);
```
The string to use for this output state.

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

\texttt{xfail(string);}

string The string to use for this output state.

8.3.1.22 Set_warning_threshold Procedure
Sets the value of \texttt{warning_threshold}. A value of 0 disables it: calls to \texttt{warning} will not turn a \texttt{PASS} or \texttt{FAIL} into an \texttt{UNRESOLVED}.

\texttt{set\_warning\_threshold(threshold);}

threshold This is the value of the new warning threshold.

8.3.1.23 Get_warning_threshold Procedure
Returns the current value of \texttt{warning\_threshold}. The default value is 3. This value controls how many \texttt{warning} procedures can be called before becoming \texttt{UNRESOLVED}.

\texttt{get\_warning\_threshold();}

8.3.1.24 Warning Procedure
Declares detection of a minor error in the test case itself. \texttt{warning} writes in the log files a message beginning with WARNING, appending the argument \texttt{string}. Use \texttt{warning} rather than \texttt{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 \texttt{number} is supplied, then this is used to set the internal count of warnings to that value.

As a side effect, \texttt{warning\_threshold} or more calls to \texttt{warning} in a single test case also changes the effect of the next \texttt{pass} or \texttt{fail} command: the test outcome becomes \texttt{UNRESOLVED} since an automatic \texttt{PASS} or \texttt{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 \texttt{UNRESOLVED}. This can be used for errors with no known side effects.

\texttt{warning(string number );}

string

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

8.3.1.25 Perror Procedure
Declares a severe error in the testing framework itself. \texttt{perror} writes in the log files a message beginning with ERROR, appending the argument \texttt{string}.

As a side effect, \texttt{perror} also changes the effect of the next \texttt{pass} or \texttt{fail} command: the test outcome becomes \texttt{UNRESOLVED}, since an automatic \texttt{PASS} or \texttt{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.

```c
perror(string number);
```

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

### 8.3.1.26 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`.

```c
note(string);
```

**string**  The string to use for this note.

### 8.3.1.27 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.

```c
untested(string);
```

**string**  The string to use for this output state.

### 8.3.1.28 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).

```c
unresolved(string);
```

**string**  The string to use for this output state.

### 8.3.1.29 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.

```c
unsupported(string);
```

**string**  The string to use for this output state.

### 8.3.1.30 Init_testcounts Procedure

```c
init_testcounts();
```
8.3.1.31 Incr_count Procedure

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

name

args

8.3.1.32 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.

```lisp
transform(toolname);
```

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

8.3.1.33 Check_conditional_xfail Procedure

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

```lisp
"[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.

```lisp
check_conditional_xfail(message targets includes excludes);
```

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

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

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

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

Specifying the conditional xfail data

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

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

8.3.1.34 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);

clear_xfail(config);
config The configuration triplets to clear.

8.3.1.35 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 --x argument to log the test results into a parsable XML file. Use the optional --n argument to print string without a trailing newline. Use the optional -- argument if string begins with ":".

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

8.3.1.36 Load_lib Procedure

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

load_lib(filespec);
filespec  The name of the DejaGnu library file to load.

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

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

8.3.2 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.

8.3.2.1 Call_remote Procedure
    call_remote(type proc dest args);
proc
dest
args

8.3.2.2 Check_for_board_status Procedure
    check_for_board_status(variable);
variable

8.3.2.3 File_on_build Procedure
    file_on_build(op file args);
op
file
args

8.3.2.4 File_on_host Procedure
    file_on_host(op file args);
op
file
args
8.3.2.5 Local_exec Procedure

local_exec(commandline inp outp timeout);

inp
outp
timeout

8.3.2.6 Remote_binary Procedure

remote_binary(host);

host

8.3.2.7 Remote_close Procedure

remote_close(shellid);

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

8.3.2.8 Remote_download Procedure

remote_download(dest file args);

dest
file
args

8.3.2.9 Remote_exec Procedure

remote_exec(hostname program args);

hostname
program
args

8.3.2.10 Remote_expect Procedure

remote_expect(board timeout args);

board
timeout
args

8.3.2.11 Remote_file Procedure

remote_file(dest args);

dest
args
8.3.2.12 Remote ld Procedure

\[ \text{remote ld(} dest \text{ prog)\}; } \\
\text{dest} \\
\text{prog} \\

8.3.2.13 Remote load Procedure

\[ \text{remote load(} dest \text{ prog args)\}; } \\
\text{dest} \\
\text{prog} \\
\text{args} \\

8.3.2.14 Remote open Procedure

\[ \text{remote open(} type)\}; } \\
\text{type} \\
\text{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 file id field in the target_info array.} \\

8.3.2.15 Remote pop conn Procedure

\[ \text{remote pop conn(} host)\); } \\
\text{host} \\

8.3.2.16 Remote push conn Procedure

\[ \text{remote push conn(} host)\); } \\
\text{host} \\

8.3.2.17 Remote raw binary Procedure

\[ \text{remote raw binary(} host)\); } \\
\text{host} \\

8.3.2.18 Remote raw close Procedure

\[ \text{remote raw close(} host)\); } \\
\text{host} \\

8.3.2.19 Remote raw file Procedure

\[ \text{remote raw file(} dest \text{ args)\}; } \\
\text{dest} \\
\text{args}
8.3.2.20 remote_raw_ld Procedure
   remote_raw_ld(dest prog);
dest
prog

8.3.2.21 Remote_raw_load Procedure
   remote_raw_load(dest prog args);
dest
prog
args

8.3.2.22 Remote_raw_open Procedure
   remote_raw_open(args);
args

8.3.2.23 Remote_raw_send Procedure
   remote_raw_send(dest string);
dest
string

8.3.2.24 Remote_raw_spawn Procedure
   remote_raw_spawn(dest commandline);
dest
commandline

8.3.2.25 Remote_raw_transmit Procedure
   remote_raw_transmit(dest file);
dest
file

8.3.2.26 Remote_raw_wait Procedure
   remote_raw_wait(dest timeout);
dest
timeout

8.3.2.27 Remote_reboot Procedure
Return value of this function depends on actual implementation of reboot that will be used, in practice it is expected that remote_reboot returns 1 on success and 0 on failure.
   remote_reboot(host);
host
8.3.2.28 Remote_send Procedure
   remote_send(dest string);
dest
string

8.3.2.29 Remote_spawn Procedure
   remote_spawn(dest commandline args);
dest
commandline
args

8.3.2.30 Remote_swap_conn Procedure
   remote_swap_conn(host);

8.3.2.31 Remote_transmit Procedure
   remote_transmit(dest file);
dest
file

8.3.2.32 Remote_upload Procedure
   remote_upload(dest srcfile arg);
dest
srcfile
arg

8.3.2.33 Remote_wait Procedure
   remote_wait(dest timeout);
dest
timeout

8.3.2.34 Standard_close Procedure
   standard_close(host);
host

8.3.2.35 Standard_download Procedure
   standard_download(dest file destfile);
dest
file
destfile
8.3.2.36 Standard_exec Procedure
    standard_exec(hostname args);

hostname
args

8.3.2.37 Standard_file Procedure
    standard_file(dest op args);

dest
op
args

8.3.2.38 Standard_load Procedure
    standard_load(dest prog args);

dest
prog
args

8.3.2.39 Standard_reboot Procedure
It looks like that this procedure is never called, instead ${{board}}_reboot defined in 'base-config.exp' will be used because it has higher priority and 'base-config.exp' is always imported by runtest.
    standard_reboot(host);

host

8.3.2.40 Standard_send Procedure
    standard_send(dest string);

dest
string

8.3.2.41 Standard_spawn Procedure
    standard_spawn(dest commandline);

dest
commandline

8.3.2.42 Standard_transmit Procedure
    standard_transmit(dest file);

dest
file
8.3.2.43 Standard_upload Procedure

    standard_upload(dest srcfile destfile);

dest
srcfile
destfile

8.3.2.44 Standard_wait Procedure

    standard_wait(dest timeout);

dest
timeout

8.3.2.45 Unix_clean_filename Procedure

    unix_clean_filename(dest file);

dest
file

8.3.3 Procedures For Using Utilities to Connect
telnet, rsh, tip, kermit

8.3.3.1 telnet Procedure

    telnet(hostname port);
    rlogin(hostname);

8.3.3.2 rsh Procedure

    rsh(hostname);

hostname This refers to the IP address or name (for example, an entry in ‘/etc/hosts’) for this target. The procedure names reflect the Unix utility used to establish a connection. The optional port is used to specify the IP port number. The value of the netport field in the target_info array is used. (was $netport)

This value has two parts, the hostname and the port number, separated by a :. If host or target is used in the hostname field, than the config array is used for all information.

8.3.3.3 Tip Procedure

    tip(port);

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

kermit(port bps);

port       Connect using the program kermit. \texttt{Port} is the device name, e.g. `/dev/ttyb`.
bps        \texttt{bps} is the line speed to use (in its per second) for the connection. The value of the \texttt{serial} field in the \texttt{target_info} array is used. (was $\texttt{serialport}$) If \texttt{host} or \texttt{target} is used in the \texttt{port} field, than the \texttt{config} array is used for all information. the \texttt{config} array is used for all information.

8.3.3.5 kermit_open Procedure

kermit_open(dest args);

dest

args

8.3.3.6 Kermit_command Procedure

kermit_command(dest args);

dest

args

8.3.3.7 Kermit_send Procedure

kermit_send(dest string args);

dest

string

args

8.3.3.8 Kermit_transmit Procedure

kermit_transmit(dest file args);

dest

file

args

8.3.3.9 Telnet_open Procedure

telnet_open(hostname args);

hostname

args

8.3.3.10 Telnet_binary Procedure

telnet_binary(hostname);

hostname
8.3.3.11 Telnet_transmit Procedure

telnet_transmit(dest file args);

dest
file
args

8.3.3.12 Tip_open Procedure

tip_open(hostname);

hostname

8.3.3.13 Rlogin_open Procedure

rlogin_open(arg);

arg

8.3.3.14 Rlogin_spawn Procedure

rlogin Spawn(dest cmdline);

dest
cmdline

8.3.3.15 Rsh_open Procedure

rsh_open(hostname);

hostname

8.3.3.16 Rsh_download Procedure

rsh_download(desthost srcfile destfile);

desthost
srcfile
destfile

8.3.3.17 Rsh_upload Procedure

rsh_upload(desthost srcfile destfile);

desthost
srcfile
destfile

8.3.3.18 Rsh_exec Procedure

rsh_exec(boardname cmd args);

boardname
cmd
args
8.3.3.19 Ftp_open Procedure

ftp_open(host);

host

8.3.3.20 Ftp_upload Procedure

ftp_upload(host remotefile localfile);

host
remotefile
localfile

8.3.3.21 Ftp_download Procedure

ftp_download(host localfile remotefile);

host
localfile
remotefile

8.3.3.22 Ftp_close Procedure

ftp_close(host);

host

8.3.3.23 Tip_download Procedure

tip_download(spawnid file);

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

file  This is the filename to download.

8.3.4 Procedures For Target Boards

8.3.4.1 Default_link Procedure

default_link(board objects destfile flags);

board
objects
destfile
flags

8.3.4.2 Default_target_assemble Procedure

default_target_assemble(source destfile flags);

source
destfile
flags
8.3.4.3 default_target_compile Procedure

    default_target_compile(source destfile type options);

source
destfile
type
options

8.3.4.4 Pop_config Procedure

    pop_config(type);

type

8.3.4.5 Prune_warnings Procedure

    prune_warnings(text);

text

8.3.4.6 Push_build Procedure

    push_build(name);

name

8.3.4.7 push_config Procedure

    push_config(type name);

type

name

8.3.4.8 Reboot_target Procedure

    reboot_target();

8.3.4.9 Target_assemble Procedure

    target_assemble(source destfile flags);

source
destfile
flags

8.3.4.10 Target_compile Procedure

    target_compile(source destfile type options);

source
destfile
type
options
8.3.5 Target Database Procedures

8.3.5.1 Board_info Procedure

    board_info(machine op args);

machine
op
args

8.3.5.2 Host_info Procedure

    host_info(op args);

op
args

8.3.5.3 Set_board_info Procedure

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

    set_board_info(entry value);

entry   The name of a board_info field to operate on.
value   The value to set the field to.

8.3.5.4 Add_board_info Procedure

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

    add_board_info(entry value);

entry   The name of a board_info field to operate on.
value   The value to add to the field.

8.3.5.5 Set_currtarget_info Procedure

    set_currtarget_info(entry value);

entry
value

8.3.5.6 Target_info Procedure

    target_info(op args);

op
args

8.3.5.7 Unset_board_info Procedure

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

    unset_board_info(entry);

entry   The name of a board_info field to operate on.
8.3.5.8 Unset_currtarget_info Procedure

unset_currtarget_info(entry);

entry

8.3.5.9 Push_target Procedure

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

push_target(name);

name The name of the target to make current connection.

8.3.5.10 Pop_target Procedure

This unsets the current target connection.

pop_target();

8.3.5.11 List_targets Procedure

This lists all the supported targets for this architecture.

list_targets();

8.3.5.12 Push_host Procedure

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

push_host(name);

name

8.3.5.13 Pop_host Procedure

This unsets the current host connection.

pop_host();

8.3.5.14 Compile Procedure

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

compile(file);

file

8.3.5.15 Archive Procedure

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

archive(file);

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

ranlib(file);
file

8.3.5.17 ExecuteAnywhere Procedure
This executes the cmd line 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

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

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

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

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

8.3.6.1 ${tool}_start Procedure
Starts a particular tool. For an interactive tool, ${tool}_start starts and initializes the tool, leaving the tool up and running for the test cases; an example is gdb_start, the start function for GDB. For a batch oriented tool, ${tool}_start is optional; the recommended convention is to let ${tool}_start run the tool, leaving the output in a variable called comp_output. Test scripts can then analyze $comp_output to determine the test results. An example of this second kind of start function is gcc_start, the start function for GCC.

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

${tool}_start();
8.3.6.2 \${tool}_load Procedure

Loads something into a tool. For an interactive tool, this conditions the tool for a particular
test case; for example, \texttt{gdb\_load} loads a new executable file into the debugger. For batch
oriented tools, \texttt{\{tool\}_load} may do nothing—though, for example, the GCC support uses
\texttt{gcc\_load} to load and run a binary on the target environment. Conventionally, \texttt{\{tool\}_load}
leaves the output of any program it runs in a variable called \texttt{$\text{exec\_output}$}. Writing
\texttt{\{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
\texttt{\{tool\}_load}.

\texttt{\{tool\}_load();}

8.3.6.3 \texttt{\{tool\}_exit Procedure}

Cleans up (if necessary) before DejaGnu exits. For interactive tools, this usually ends the
interactive session. You can also use \texttt{\{tool\}_exit} to remove any temporary files left over
from the tests. \texttt{runtest} calls \texttt{\{tool\}_exit}.

\texttt{\{tool\}_exit();}

8.3.6.4 \texttt{\{tool\}_version Procedure}

Prints the version label and number for \texttt{\{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.

\texttt{\{tool\}_version();}

8.3.7 Utility Procedures

8.3.7.1 Getdirs Procedure

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

\texttt{getdirs(rootdir pattern);}

\texttt{args pattern}

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

8.3.7.2 Find Procedure

Search for files whose names match \texttt{pattern} (using shell wildcard characters for filename
expansion). Search subdirectories recursively, starting at \texttt{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.

\texttt{find(rootdir pattern);}

\texttt{rootdir} The top level directory to search the search from.

\texttt{pattern} A csh "glob" style regular expression representing the files to find.
8.3.7.3 Which Procedure

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

```
which(file);
```

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

8.3.7.4 Grep Procedure

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

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

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

*filename* The file to search.

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

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

8.3.7.5 Prune Procedure

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

8.3.7.6 Slay Procedure

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

8.3.7.7 Absolute Procedure

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

8.3.7.8 Psourc Procedure

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

8.3.7.9 Runtest_file_p Procedure

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

runtest_file_p(runtests testcase);

runtests  The list of patterns to compare against.

testcase  The test case filename.

8.3.7.10 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.

8.3.7.11 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.

8.3.7.12 unsetenv Procedure
Unsets the environment variable var.

unsetenv(var);

var  The environment variable to unset.

8.3.7.13 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.

8.3.7.14 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.
8.3.8 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.

8.3.8.1 Libgloss_link_flags Procedure
   libgloss_link_flags(args);
   args

8.3.8.2 Libgloss_include_flags Procedure
   libgloss_include_flags(args);
   args

8.3.8.3 Newlib_link_flags Procedure
   newlib_link_flags(args);
   args

8.3.8.4 Newlib_include_flags Procedure
   newlib_include_flags(args);
   args

8.3.8.5 Libio_include_flags Procedure
   libio_include_flags(args);
   args

8.3.8.6 Libio_link_flags Procedure
   libio_link_flags(args);
   args

8.3.8.7 G++_include_flags Procedure
   g++_include_flags(args);
   args

8.3.8.8 G++_link_flags Procedure
   g++_link_flags(args);
   args

8.3.8.9 Libstdc++_include_flags Procedure
   libstdc++_include_flags(args);
   args
8.3.8.10 Libstdc++_link_flags Procedure
   libstdc++_link_flags(args);
args

8.3.8.11 Get_multilibs Procedure
   get_multilibs(args);
args

8.3.8.12 Find_binutils_prog Procedure
   find_binutils_prog(name);
name

8.3.8.13 Find_gcc Procedure
   find_gcc();

8.3.8.14 Find_gcj Procedure
   find_gcj();

8.3.8.15 Find_g++ Procedure
   find_g++();

8.3.8.16 Find_g77 Procedure
   find_g77();

8.3.8.17 Find_gfortran Procedure
   find_gfortran();

8.3.8.18 Process_multilib_options Procedure
   process_multilib_options(args);
args

8.3.8.19 Add_multilib_option Procedure
   add_multilib_option(args);
args

8.3.8.20 Find_gas Procedure
   find_gas();

8.3.8.21 Find_ld Procedure
   find_ld();

8.3.8.22 Build_wrapper Procedure
   build_wrapper(gluefile);
   gluefile
8.3.8.23 Winsup_include_flags Procedure

```tcl
winsup_include_flags(args);
```

args

8.3.8.24 Winsup_link_flags Procedure

```tcl
winsup_link_flags(args);
```

args

8.3.9 Procedures for debugging your Tcl code.

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

8.3.9.1 Dumpvars Procedure

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

```tcl
dumpvars(vars);
```

vars The variables to dump.

8.3.9.2 Dumplocals Procedure

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

```tcl
dumplocals(args);
```

args

8.3.9.3 Dumprocs Procedure

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

```tcl
dumprocs(pattern);
```

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

8.3.9.4 Dumpwatch Procedure

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

```tcl
dumpwatch(pattern);
```

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

8.3.9.5 Watcharray Procedure

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

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

8.3.9.6 Watchvar Procedure

```tcl
watchvar(var type);
```
8.3.9.7 Watchunset Procedure
This breaks program execution when the variable var is unset. It is abbreviated as wu.

watchunset(arg);

8.3.9.8 Watchwrite Procedure
This breaks program execution when the variable var is written. It is abbreviated as ww.

watchwrite(var);

var The variable to watch.

8.3.9.9 Watchread Procedure
This breaks program execution when the variable var is read. It is abbreviated as wr.

watchread(var);

var The variable to watch.

8.3.9.10 Watchdel Procedure
This deletes a watchpoint from the watch list. It is abbreviated as wd.

watchdel(args);

8.3.9.11 Print Procedure
This prints the value of the variable var. It is abbreviated as p.

print(var);

var

8.3.9.12 Quit Procedure
This makes runtest exit. It is abbreviated as q.

quit();

8.4 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
9 Unit Testing API

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

9.1.1 Pass Function
This prints a message for a successful test completion.

```c
pass(msg);
```

9.1.2 Fail Function
This prints a message for an unsuccessful test completion.

```c
fail(msg);
```

9.1.3 Untested Function
This prints a message for an test case that isn’t run for some technical reason.

```c
untested(msg);
```

9.1.4 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.

```c
unresolved(msg);
```

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

```c
totals();
```

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

9.2.1 Pass Method
This prints a message for a successful test completion.

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

9.2.2 Fail Method
This prints a message for an unsuccessful test completion.

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

9.2.3 Untested Method
This prints a message for an test case that isn’t run for some technical reason.

```c
TestState::untested(msg);
```
9.2.4 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);
```

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

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