Howto: Using *DejaGnu* for Testing

A Simple Introduction

Jeremy Bennett
Embecosm

Application Note 8. Issue 1
Publication date April 2010

Table of Contents

List of Tables

Chapter 1. Introduction

DejaGnu is a testing framework, originally developed for the GNU project. This application note was written in response to the author's frustration at setting up a *DejaGnu* test framework for the first time, using the existing documentation [3].

DejaGnu can be used standalone, but it is most useful when integrated with the GNU autotools (*autoconf automake* and *libtool*). This application note describes both modes of use.

This application note is deliberately incomplete. It only covers the most commonly used features and options. The intention is that, having got an initial test suite up and running, the user will be confident in reading the full *DejaGnu* documentation (see Section 1.2).

1.1. Target Audience

Setting up *DejaGnu* for the first time is a challenge. This application note is intended for software engineers putting together their first test suite using *DejaGnu*.

1.2. Further information

The main source of information is the *DejaGnu* user guide by Rob Savoye [3]. The most useful section is the *Unit Testing API*, which lists all the procedures which can be used when writing tests.

DejaGnu is implemented as the **runtest** command. It is documented in its own manual page (**man runtest**). However the manual page is incomplete, and additional options to this command may be identified by using **runtest --help**.

Tests are written in an extension to the Tool Command Language (TCL), known as *expect*. TCL is extensively documented on its website (see www.tcl.tk/doc). Of particular value is the TCL tutorial on that site [1].

Expect has its own manual page (**man expect**), which provides a "succinct" description of the main language commands. Comprehensive documentation may be found in the book *Exploring Expect* by Don Libes [2].

1.3. About Embecosm Application Notes

Embecosm publishes a series of free and open source application notes, designed to help working engineers with practical problems.

Feedback is always welcome, which should be sent to `<info@embecosm.com>`.

Chapter 2. Overview of *DejaGnu*

DejaGnu is designed to run on POSIX compliant systems and is compliant with the POSIX 1003.3 standard for test frameworks. It comprises a command to run tests (**runtest**), a language in which to write tests (*expect*), a standard directory structure for the tests and a set of configuration files.

Note
DejaGnu is not installed by default on most Linux systems. However most systems provide it as a standard extra. If not, it can be downloaded from www.gnu.org/software/dejagnu.

2.1. The runtest command

Central to *DejaGnu* is the **runtest** command. It takes the general form:

```
runtest [options] [test]
```

The most important options when getting started are as follows.

`--tool` *`toolname`*	The naming of this option is confusing. It is the name of the group of tests being carried out. This might be the name of a tool (for example **`or1ksim`**), but it might equally be an indicator of a type of test (for example **`unit`**). The *`toolname`* is important, because it is used to construct the names of other entities: the directories where tests may be found and the name of a tool specific configuration file.
`--srcdir` *`dir`*	This specifies a path where directories of tests may be found. A directory containing tool specific configuration files (**`lib`**) is also found in this directory. The actual names of the directories containing tests must be prefixed by the name of the tool, and by convention end with the suffix **`.tests`**. Thus for the tool **`or1ksim`**, the tests might appear in directories named **`or1ksim.tests`** or **`or1ksim-profiling.tests`**.
`--outdir` *`dir`*	This specifies a directory where all the output logs will be placed.
`--target_board` *`board`*	This is one of the hidden options, only shown with **runtest □□help**. It specifies board(s) on which the tests should be run. This also controls which board specific configuration file(s) will be used (see Section 2.4).

If no *`test`* is specified, **runtest** will run all the tests it can find, which match the specified *`toolname`*. Otherwise it will run only the test specified.

Note
Options may also be set in configuration files (see Section 2.4). Where an option is set both on the command line and in a configuration file, the command line will take precedence.

Caution
There are a number of options which are not documented in the manual. You can see all of the options by running **runtest --help**

2.2. The expect language

Strictly speaking, *expect* is a tool, rather than a language. However it is easiest to regard it as a language which extends TCL. Before writing code using *expect* you should be fully familiar with TCL [1].

Expect scripts spawn test programs, to which they supply input and check that the response is as expected. A full description of how to write tests is given in Chapter 3 below.

2.3. A typical test framework directory structure

DejaGnu has some basic expectations of where it will find files, although this can be overridden with command line options to **runtest**.

By convention, the test framework lives within a directory **testsuite**. Within that directory are the following sub-directories:

config	An optional directory containing *expect* configuration files for each of the different types of target board that might be tested (see Section 2.4). The use of this directory depends on it being set up as part of the configuration. These have the general name ***board*.exp**. For example **unix.exp** for the default Unix target.
lib	A directory containing *expect* configuration files for each of the tools being tested (see Section 2.4). These files are named ***toolname*.exp**, where *toolname* is the tool being tested.
Test directories	Directories named ***toolname*[*type*].tests** contain *expect* tests for the tool *toolname*. The optional *type* field allows the tests to be split into several directories. For example a compiler tool named **xcc** might have tests both of the compiler (in one directory, **xcc-compile.tests**) and the resulting code (in a second directory, **xcc-execute.tests**). Strictly speaking the suffix **.tests** is not required, but it is a common convention. Within each test directory, the tests, written in *expect*, all have the file suffix **.exp**.

2.4. DejaGnu configuration files

Configuration files are used to set global variables and to define helper functions.

DejaGnu has a number of global variables which are set to correspond to **runtest** command line arguments. However they can alternatively be set in the various configuration files (if both are used, the command line takes precedence). The *DejaGnu* guide [3] documents them in full, but some of the most useful are shown in in Table 2.1.

runtest	***expect***	**Description**
--all	**all_flag**	Display all test results if set (default is only to display failures).
--srcdir	**srcdir**	The directory containing test sub-directories and the tool specific configuration directory, **lib**.
--tool	**tool**	The name of the tool being tested.
--verbose	**verbosity**	The degree of verbosity required. Default is 0. Value of the variable is incremented for each time it appears on the command line.

Table 2.1. runtest options and corresponding *expect* variables.

DejaGnu has a plethora of configuration files, each of which contain *expect* script that is processed before each test. The historical naming of some of these files can be confusing.

Although these configuration files are all optional, **runtest** may complain if it can't find them. So it's worth setting them up, just to stop the messages.

The configuration files are read in the following sequence.

Personal config file
: This is in the user's home directory, **`~/.dejagnurc`**. In general it is only used to set variables that control personal taste in output.

Local config file
: This is always named **`site.exp`** and is looked for in the directory where the tests are being run. In general it is used to set variables specific to the tool or tools being tested.
If GNU autotools (*autoconf*, *automake* and *libtool*) are being used to drive *DejaGnu* this file will be created automatically whenever **make** is run. It will be placed in the main test suite directory, but within the build directory hierarchy, *not* the source directory hierarchy. It is this directory from which **runtest** will be executed.

When GNU autotools generate **`site.exp`**, the file is created in two halves, separated by a comment line. Only the first part is regenerated every time **make** is run, so changes may safely be made in the second half.

This is a convenient place for quickly testing changes. However the local config file in an autotools environment is inherently transient (it is in the build directory, not the source directory), so permanent changes should be placed in one of the other config files.

Global config file
: This is intended as a file to allow a group testing the same product to share common settings. It is identified by the **`DEJAGNU`** environment variable, which should always be set to avoid warnings.

Board config file
: This is used for settings specific to a particular target board, as specified by the `--target_board` option to **runtest** (see Section 2.1). If no target board is specified, then *DejaGnu* running on a POSIX system will typically decide the target is *Unix* and look for a file named **`unix.exp`**.
The search path for board configuration files is in the global variable **`boards_dir`**, which by default includes **`/usr/share/dejagnu/baseboards`** and **`/usr/share/dejagnu/config`**.

It is common to append additional directories to this search path in one of the earlier configuration directories (typically the global config file) to provide additional board specific information. This can be done by using the TCL **`lappend`** function. For example, to look in the **`config`** sub-directory of the main test directory for board specific settings, the following would be suitable in the global configuration file.

```
lappend boards_dir "$srcdir/boards"
```

Note
If multiple board configuration files are found in the different directories, they will all be used in the order they are found.

Tool config file
: Finally *DejaGnu* will load code from a tool specific configuration file. This will be found in the **`lib`** sub-directory of the main test directory. For a tool named *`toolname`*, the file will be **`lib/`*`toolname`*`.exp`**.

This is the place to put helper functions for use by test code.

Note

When running **runtest**, multiple test programs may be found in the test directory or directories. These are effectively concatenated, so they are run one after the other after running all the configurations.

The important thing to remember is that this means the configuration files are only read once, *not* once for each test.

Chapter 3. Writing tests using *expect*

Each *DejaGnu* test is a sequence of *expect* commands. These are standard TCL commands with some additions, which make the language particularly suitable for testing. **man expect** documents all the commands specific to *expect*. The most useful of these are:

`expect`	This is the most important of the commands. It takes a series of pairs of patterns and actions and waits until one of the patterns matches the output of a spawned process (see **`spawn`** below), or a specified time period has passed or an end of file has been seen. When a pattern matches, its corresponding body is executed. This command has a considerable number of options, and is described in detail in a separate section (see Section 3.2).
`send`	This command is used to send data to the standard input of a spawned process (see **`spawn`** below).
`send_error` `send_log` `send_user`	These commands are used to send output to respectively the standard error of the user, the log file and the standard output of the user. In other words for data that is *not* to go to the spawned process.

`spawn`

```
spawn [opts] program args
```

`spawn` starts a program and its arguments in a child process, connecting its stdin, stdout and stderr so they may be written and read by other *expect* commands, most notably the **`expect`** command.

Note

There is the potential for confusion here, since *expect* is both the name of the language and a command within that language.

Expect defines and uses a number of TCL global variables. The most important are associated with the **`expect`** command and documented in the section on that command (see Section 3.2).

On top of this *DejaGnu* defines a number of procedures which facilitate testing. These are documented in the *DejaGnu* manual's section on Unit Testing [3]. The most useful are:

`fail` `pass` `xfail` `xpass` `untested` `unresolved` `unsupported`	These procedures all report the result of a test. They are described in more detail below (see Section 3.1.
`warning`	Writes a message to the log, prepended by the string **`WARNING`**. Once more than **`warning_threshold`** warnings have been given the test is assumed to be unresolved. The next call to one of the result procedures will behave as though **`unresolved`** had been called. The value of **`warning_threshold`** (default 3) may be read and written with **`get_warning_threshold`** and **`set_warning_threshold`**.

`perror`	Writes a message to the log, prepended by the string **`ERROR`**. The next call to one of the result procedures will behave as though **`unresolved`** had been called.

3.1. Test results

The results from all the tests encountered by **`runtest`** are reported in the log file and counted, for reporting on completion. The results are triggered by calling the appropriate result procedure, which takes a string to identify the test concerned. The possible results are as follows:

FAIL

Indicates that a test has failed. Triggered by calling the **`fail`** procedure.

PASS

Indicates that a test has passed. Triggered by calling the **`pass`** procedure.

XFAIL

Indicates that a test has failed as expected. Triggered by calling the **`xfail`** procedure.

Note

POSIX 1003.3 does not recognize the concept of expected failure (tests either fail or pass), so compliant tests should not generate this result.

XPASS

Indicates that a test which was expected to fail has passed. Triggered by calling the **`xpass`** procedure.

Note

POSIX 1003.3 does not recognize the concept of expected failure (tests either fail or pass), so compliant tests should not generate this result.

UNTESTED

Indicates that a feature was not tested. Triggered by calling the **`untested`** procedure. This is a good way to mark tests that need to be completed.

UNRESOLVED

Indicates that the output from a test needs manual inspection. Triggered by calling the **`unresolved`** procedure. Commonly used when tests timeout.

Note

Any test which calls **`perror`** or calls **`warning`** more than **`warning_threshold`** times will cause the next call to any result function in this list to behave as though **`unresolved`** was called.

UNSUPPORTED

Indicates that a test is not supported. Triggered by calling the **`unsupported`** procedure. Used for tests which depend on some conditionally available feature. For example for tests which cannot run on a particular target board.

3.2. The `expect` command

The general form of this command is

```
expect -flags pat1 body1 ... -flags patn bodyn
```

The command waits until one of the patterns matches the output of a spawned process, a specified time period has passed, or an end-of-file is seen. It then executes the corresponding body.

If (as is usual) the command takes more than one line, the arguments must be surrounded by braces. However substitutions will still occur within these braces, unlike standard TCL

A very simple example might be as follows.

```
expect {
    ERROR            {fail "Error encountered.\n"; }
    "Test complete"  {pass "Test completed.\n";}
    timeout          {unresolved "Timeout.\n";}
}
```

The **`expect`** command returns a result, which is the result of the body executed on a pattern match.

The **`expect`** command will be called each time there is new input. It is important to understand two aspects of **`expect`** command behavior.

1. The **`expect`** command will skip past any unmatched text. So the above example would pass if the spawned program generated the following output.

   ```
   Computing results
   Final result is 42
   Test complete
   ```

   The first two lines don't match, so are ignored.

2. The **`expect`** command is *not* line oriented. Thus the following output from the spawned program would also pass.

   ```
   FooTest completeBar
   ```

3.2.1. Patterns for use with the `expect` command

There are a wide range of possible patterns that *expect* can use.

Plain strings	Plain strings. In the previous example the strings **`ERROR`** and **`"Test complete"`** were used. Double quotation marks are needed where the string contains a space.
`eof`	Matches if the stream from the spawned process reaches end of file. Typically because the spawned process has completed execution.
`timeout`	Matches if more than **`timeout`** seconds have passed since the spawned process started execution. The default timeout period is 10 seconds, but may be changed by setting the **`timeout`** global variable. The timeout value may be also set for the current **`expect`** command by using the **`-timeout`** flag as a pattern (see below).
`default`	Equivalent to matching either **`eof`** or **`timeout`**.

	-timeout *t*. Use *t* seconds as the timeout for the current **expect** command.
full_buffer	By default, the **expect** command buffers up to 2000 bytes. If more than this is encountered while reading bytes, earlier bytes will be forgotten. This keyword will match if the buffer is full, allowing this circumstance to be trapped. **Note** The size of the buffer can be increased using the procedure **match_max**. However very large values down will slow down the pattern matcher.
null	Matches a single ASCII character 0.
"Globbed" expressions	These are patterns specified as for the TCL **string match** command, which is similar to the syntax of shell regular expressions (commonly known as "glob" patterns). It is possible that patterns might match flags to the **expect** command, in which case they can be protected by using the **-gl** flag. So for example the following would match the string **"-gl"**.

```
expect {
    -gl -gl {puts "Matched -gl\n"}
}
```

In general any pattern starting with '-' should be protected using **-gl** to future proof against new flags.

Regular expressions	These patterns follow the syntax defined by the TCL **regexp** command. They are introduced with the flag **-re**.
Exact strings	These are prefixed by the **-ex** flag. This is needed for strings containing characters such as '*' that would otherwise be interpreted as part of a globbed pattern.
Case insensitive match	Prefixing any pattern by the **-nocase** flag will cause the input to be matched as though it were all lower case. The pattern should thus be all in lower case.

When matching a pattern, any matching and previously unmatched output is saved in the variable **expect_out(buffer)**. The matched output may be found in **expect_out(0,string)**.

If a regular expression with sub-expressions was used, then the matching sub-expressions (up to 9 in total) may be found in **expect_out(1,string)** through **expect_out(1,string)**.

If the flag **-indices** was used before any match, the start and end indices of the matching string can be found in **expect_out(0,start)** and **expect_out(0,end)**. Where the pattern was a regular expression, start and end positions of up to 9 sub-strings may be found in **expect_out(1,start)** and **expect_out(1,end)** through **expect_out(9,start)** and **expect_out(9,end)**.

These strings are useful when evaluating the body associated with the pattern matched.

Once matching is complete, the matched output (and any preceding output) is discarded from the internal buffers. However it can be retained by preceding the match with the **-notransfer** flag. This has little value in finished scripts, but can help when developing and debugging tests.

3.3. An example test

Writing a test is a matter of spawning the test program, then using the **expect** to match the output and report failure as appropriate.

A simple example is as follows.

```
# Timeout reduced to 3 seconds
set timeout 3

# The name of this test and the command we will run
set test_name "Simple test"
set command_line "or32-elf-sim -f default.cfg test-prog.or32"

# When verbose, tell the user what we are running
if { $verbose > 1 } {
    send_user "starting $command_line\n"
}

# Run the program.
spawn $command_line

expect {
    # Check for any warning messages in the output first
    Warning {
        fail "$test_name: warning: $expect_out(buffer)"
    }

    # Check for any error messages
    ERROR {
        fail "$test_name: error: $expect_out(buffer)"
    }

    # The string indicating successful completion
    "Test complete" {
         pass "$test_name\n"
    }

    # EOF and timeout only come after everything else. EOF must be an error.
    eof {
        fail "$test_name: EOF\n"
    }

    # Timeout requires inspection to determine the cause of failure.
    timeout {
        unresolved "$test_name: timeout"
    }
}
```

This test will look first for the strings **"Warning"** or **"ERROR"** in the output stream from the spawned program. Then it will look for the string **Test complete**. Finally it will check for end-of-file or timeout.

The ordering matters. For a small program, the spawned output could all fit in the buffer. By that time, the program might have completed execution. So that buffer would match both **`Test complete`** and end-of-file. So we must check for successful completion first.

Similarly the program might generate warnings or errors, but still print **`Test complete`**. So we should check for warnings or errors *before* we test for successful completion.

3.3.1. Automating testing

The example in Section 3.3 is a fairly common framework, and it makes sense to automate it using TCL procedure. For example the test name and command line could be passed as arguments to the procedure as follows.

```
proc runmytest { test_name command_line } {
    global verbose

    # When verbose, tell the user what we are running
    if { $verbose > 1 } {
        send_user "starting $command_line\n"
    }

    # Run the program.
    spawn $command_line

    expect {
        # Check for any warning messages in the output first
        Warning {
        fail "$test_name: warning: $expect_out(buffer)"
        }

        # Check for any error messages
        ERROR {
        fail "$test_name: error: $expect_out(buffer)"
        }

        # The string indicating successful completion
        "Test complete" {
             pass "$test_name\n"
        }

        # EOF and timeout only come after everything else. EOF must be an error.
        eof {
        fail "$test_name: EOF\n"
        }

        # Timeout requires inspection to determine the cause of failure.
        timeout {
            unresolved "$test_name: timeout"
        }
    }
}
```

The code is identical, except we must note from within the procedure that **`verbose`** is a global variable.

Our series of tests could then just be as follows

```
# Timeout reduced to 3 seconds
set timeout 3

runmytest "Simple test" "or32-elf-sim -f default.cfg test-prog.or32"
runmytest "Harder test" "or32-elf-sim -f default.cfg test-prog2.or32"
runmytest "Hardest test" "or32-elf-sim -f default.cfg test-prog3.or32"
```

The correct place for a procedure shared amongst a number of tests like this is the tool configuration file (see Section 2.4).

Chapter 4. Using *DejaGnu* standalone

To run a *DejaGnu* test framework standalone is straightforward.

1. Create a directory structure for the test framework (see Section 2.3).
2. Write any test programs and compile them.
3. Write tests in *expect* and place them in the tool specific test directory (or directories) (see Chapter 3)
4. Place any TCL support procedures for the tests in the tool specific configuration file (see Section 2.4 and Section 3.3.1).
5. Run **runtest**, specifying the source directory and tool name.

This sequence of steps can easily be captured in either a script file, or a hand-written **make** file.

Chapter 5. Using *DejaGnu* with *autoconf*, *automake* and *libtool*

Almost all use of *DejaGnu* is in practice within the context of projects using the GNU autotools (*autoconf*, *automake* and *libtool*). These tools understand *DejaGnu* and provide specific hooks.

The overall directory structure is as described earlier (see Section 2.3). By convention the tests are all run from a directory named **`testsuite`**, but this need not be the case. That directory, as well as containing the tool specific test directories, tool specific configuration directory (**`lib`**) and possible a board specific configuration directory should also contain the global configuration file and directories for any test programs that must be compiled.

Autotools programs are usually configured and built in a separate directory structure. **runtest** will be executed in the **`testsuite`** directory, within the build directory hierarchy, *not* the source hierarchy. It is within this directory that the local configuration file, **`site.exp`** will be generated when running **make**.

There are no explicit changes needed in the use of *libtool*. However some changes are needed to **`configure.ac`** for *autoconf* and **`Makefile.am`** for *automake*.

5.1. The `configure.ac` file

This is the file that is modified by *autoconf* and provides ultimately the definitions that will feed into *automake*. It will need some additional changes to tell it about *DejaGnu*

Note

In some older systems, this file may be named **`configure.in`**.

Where there are multiple configuration files, it is usually the top level one which is changed. The important thing is it should be the configuration file which feeds into the transformation of the **`Makefile.am`** file in the main test directory (**`testsuite`**).

The file should be extended to set the **`DEJAGNU`** environment variable to point to the global configuration file. There are two approaches to this. The simplest is just to set the value. However it is more flexible to only set a value if it is not already set. This allows the user to try different global configurations by manually setting **`DEJAGNU`** before running tests.

```
if test x"$DEJAGNU" = x
then
  DEJAGNU="\$(top_srcdir)/testsuite/global-conf.exp"
fi

AC_SUBST(DEJAGNU)
```

5.2. `Makefile.am` files

There should be a **`Makefile.am`** in the main test directory (**`testsuite`**) and each of its sub-directories.

If **dejagnu** appears in **AUTOMAKE_OPTIONS**, then *automake* will assume a *DejaGnu* based test system. This will be run when the user uses **make check**.

The tests must pick up the **DEJAGNU** environment variable. So it should be exported from the **Makefile.am** in the main test directory.

The tool(s) must be named. This is done by setting the **DEJATOOL** variable. If there are multiple tools to be tested, specify them all with **DEJATOOL**

The following shows part of a **Makefile.am** for a typical test main directory.

```
SUBDIRS = config          \
          lib             \
          libsim.tests    \
          or1ksim.tests   \
          test-code       \
          test-code-or1k

EXTRA_DIST = global-conf.exp

# Setup for DejaGNU
AUTOMAKE_OPTIONS    = dejagnu

export DEJAGNU

DEJATOOL            = libsim  \
                      or1ksim
```

Note the use of **EXTRA_DIST** to specify that the global configuration file must be added to the distribution. In this example there is a board specific configuration directory (**config**), an *expect* test directory for each of two tools (**libsim.tests** and **or1ksim.tests**) and two directories of test program code (**test-code** and **test-code-or1k**).

Makefile.am files are needed in the various config and *expect* test directories, but only to ensure their code is added to the distribution. There is nothing that needs compiling. For example, the **Makefile.am** in the tool specific configuration directory (**lib**) is as follows.

```
# Just distribute this stuff
EXTRA_DIST = libsim.exp   \
             or1ksim.exp
```

5.3. Fine tuning autoconf and automake with DejaGnu

The flags passed to **runtest** by default are held in the variable **RUNTESTDEFAULTFLAGS**. Its default value is **--tool $$tool --srcdir $$srcdir**. **$$tool** will expand to each tool specified in **DEJATOOL** and **$$srcdir** to the full name of the source directory.

This variable can be overridden in the configuration file.

It is also sometimes useful to override **runtest** options just within the **Makefile.am**. This can be done by setting the variable **AM_RUNTESTFLAGS** within the **Makefile.am**.

5.3.1. Example: Short test names.

DejaGnu will log the name of each file of *expect* code found in the test directory. However by default it will use the full file name, which can make for hard to read listings.

This can be adjusted by redefining **RUNTESTDEFAULTFLAGS** in **configure.ac**.

```
# The following line will ensure that short names are used for test names.
RUNTESTDEFAULTFLAGS="--tool \$\$tool"
AC_SUBST(RUNTESTDEFAULTFLAGS)
```

By not defining `--srcdir` on the command line, **runtest** will look instead in the local configuration file, automatically generated by *automake* and *autoconf*, which includes a definition of the global variable **srcdir**.

This is useful, because in the local configuration file, the **srcdir** is specified with its relative name (for example **../../testsuite**). This is shorter than the value of **$srcdir** that would be used as the command line value. The result is that the name of the tests run will be shorter, making for clearer output. Instead of:

```
                === or1ksim tests ===

Schedule of variations:
    unix

Running target unix
Using ../../testsuite/config/unix.exp as board description file for target.
Running /home/jeremy/svntrunk/Projects/or1ksim/testsuite/or1ksim.tests/basic
.exp ...
Running /home/jeremy/svntrunk/Projects/or1ksim/testsuite/or1ksim.tests/cache
.exp ...

...

Running /home/jeremy/svntrunk/Projects/or1ksim/testsuite/or1ksim.tests/mycom
press.exp ...
Running /home/jeremy/svntrunk/Projects/or1ksim/testsuite/or1ksim.tests/tick.
exp ...

                === or1ksim Summary ===

# of expected passes            21
```

We get the more concise:

```
                === or1ksim tests ===

Schedule of variations:
    unix

Running target unix
Using ../../testsuite/config/unix.exp as board description file for target.
Running ../../testsuite/or1ksim.tests/basic.exp ...
Running ../../testsuite/or1ksim.tests/cache.exp ...

...

Running ../../testsuite/or1ksim.tests/mycompress.exp ...
Running ../../testsuite/or1ksim.tests/tick.exp ...

                === or1ksim Summary ===

# of expected passes            21
```

Glossary

Expect
: *Expect* is a program that "talks" to other interactive programs. Colloquially it is used to refer to the language used by that program, which is an extension of TCL.
See also: Tool Command Language

runtest
: **runtest** is the main command of the *DejaGnu* testing framework. It runs a series of tests written in *expect* and reports the results of those tests.
See also: Expect

Tool Command Language (TCL)
: TCL is an open source dynamic scripting language, widely used for rapid prototyping and system admin tasks. It is designed to be highly extensible.

References

[1] Tcl Tutorial Clif Flynt. Version for TCL 8.5 available at www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html.

[2] Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs Don Libes. O'Reilly and Associates 1995, 1-56592-090-2.

[3] DejaGnu: The GNU Testing Framework Rob Savoye. Free Software Foundation 2004. Available at www.gnu.org/software/dejagnu/manual.