# The Megatest Users Manual

Matt Welland

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| 1.5 | June 2020 | | MW |

# Contents

# Preface

This book is organised as three sub-books; getting started, writing tests and reference.

**License**

```
Copyright 2006-2020, Matthew Welland.

This document is part of Megatest.

Megatest is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

Megatest is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with Megatest.  If not, see <http://www.gnu.org/licenses/>.
```

# Chapter 1

# Why Megatest?

Megatest was created to provide a generalized tool for managing suites of regression tests and to provide a multi-host, distributed alternative to "make". The EDA world is littered with proprietory, company-specific tools for this purpose and by going open source and keeping the tool flexible the hope is that Megatest could be useful to any team at any company for continuous integration and almost any other general automation tasks.

# Chapter 2

# Megatest Design Philosophy

Megatest is a distributed system intended to provide the minimum needed resources to make writing a suite of tests and tasks for implementing continuous build for software, design engineering or process control (via owlfs for example) without being specialized for any specific problem space. Megatest in of itself does not know what constitutes a PASS or FAIL of a test or task. In most cases megatest is best used in conjunction with logpro or a similar tool to parse, analyze and decide on the test outcome.

- Self-checking - make it as easy as possible to write self-checking tests (as opposed to using deltas, i.e. tests that compare with a previous measurement to deterine PASS/FAIL).

- Traceable - environment variables, host OS and other possibly influential variables are captured and kept recorded.

- Immutable - once a test is run it cannot be easily overwritten or modified accidentally.

- Repeatable - test results can be recreated in the future using all the original variables.

- Relocatable - the testsuite or automation area can be checked out and the tests run anywhere in the disk hierarchy.

- Encapsulated - the tests run in self-contained directories and all inputs and outputs to the process can be found in the run areas.

- Deployable - a testsuite is self-contained and can be bundled with a software project and easily used by others with little to no setup burden.

# Chapter 3

# Megatest Architecture

## 3.1 Data separation

All data to specify the tests and configure the system is stored in plain text config files. All system state is stored in an sqlite3 database.

## 3.2 Distributed Compute

Tests are launched using the launching system available for the distributed compute platform in use. A template script is provided which can launch jobs on local and remote Linux hosts. Currently megatest uses the network filesystem to call home to your master sqlite3 database. Megatest has been used with the Intel Netbatch and lsf (also known as openlava) batch systems and it should be straightforward to use it with other similar systems.

# Chapter 4

# Overview

## 4.1 Stand-alone Megatest Area

A single, stand-alone, Megatest based testsuite or "area" is sufficient for most validation, automation and build problems.

# Megatest Stand-Alone Area

## multi-area dashboard and xterm
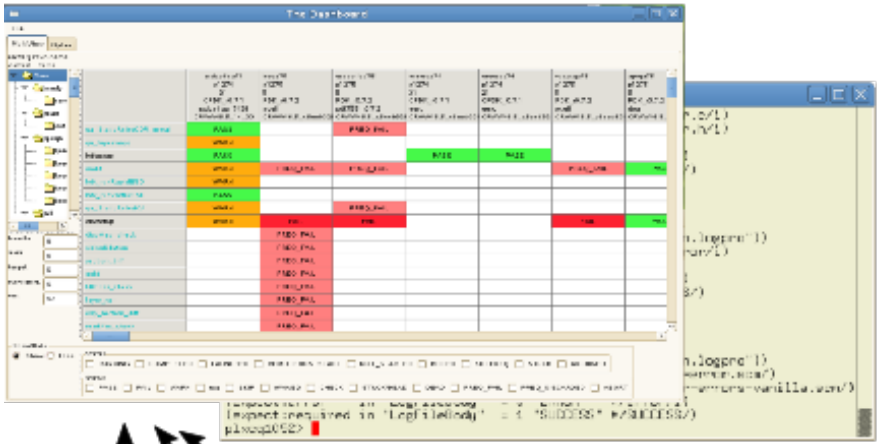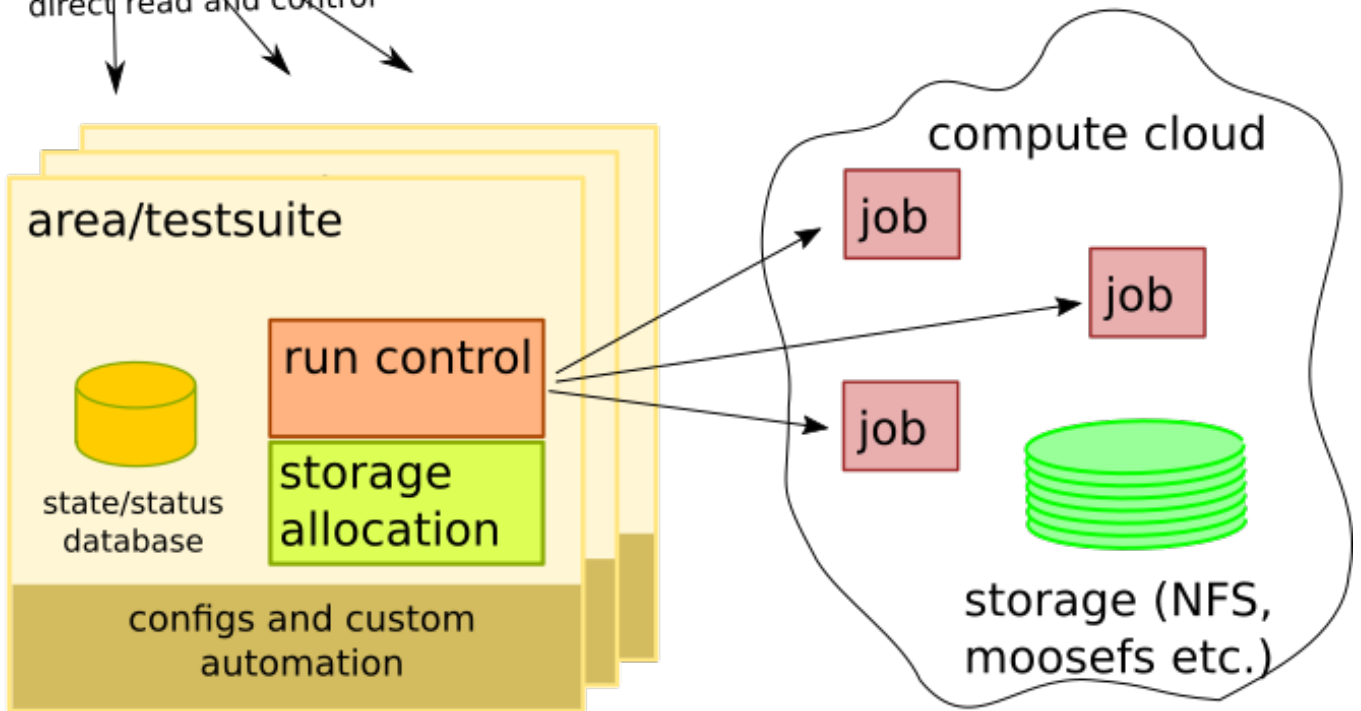


Megatest is designed as a distributed or decoupled system. This means you can run the areas stand-alone with no additional infrastructure. I.e. there are no databases, web servers or other centralized resources needed. However as your needs grow you can integrate multiple areas into a bigger system.

### 4.1.1  Component Descriptions

1. Multi-area dashboard and xterm. A gui (the dashboard) is usually the best option for controlling and launching runs but all operations can also be done from the commandline. Note: The not yet released multi-area dashboard replaces the old dashboard for browsing and controlling runs but for managing a single area the old dashboard works very well.

2. Area/testsuite. This is your testsuite or automation definition and consists of the information in megatest.config, runconfigs.config and your testconfigs along with any custom scripting that can't be done with the native Megatest features.

3. If your testsuite or build automation is too large to run on a single instance you can distribute your jobs into a compute server pool. The only current requirements are password-less ssh access and a network filesystem.

## 4.2 Full System Architecture

# Megatest Full System Architecture

multi-area dashboard and xterm            multi-area web
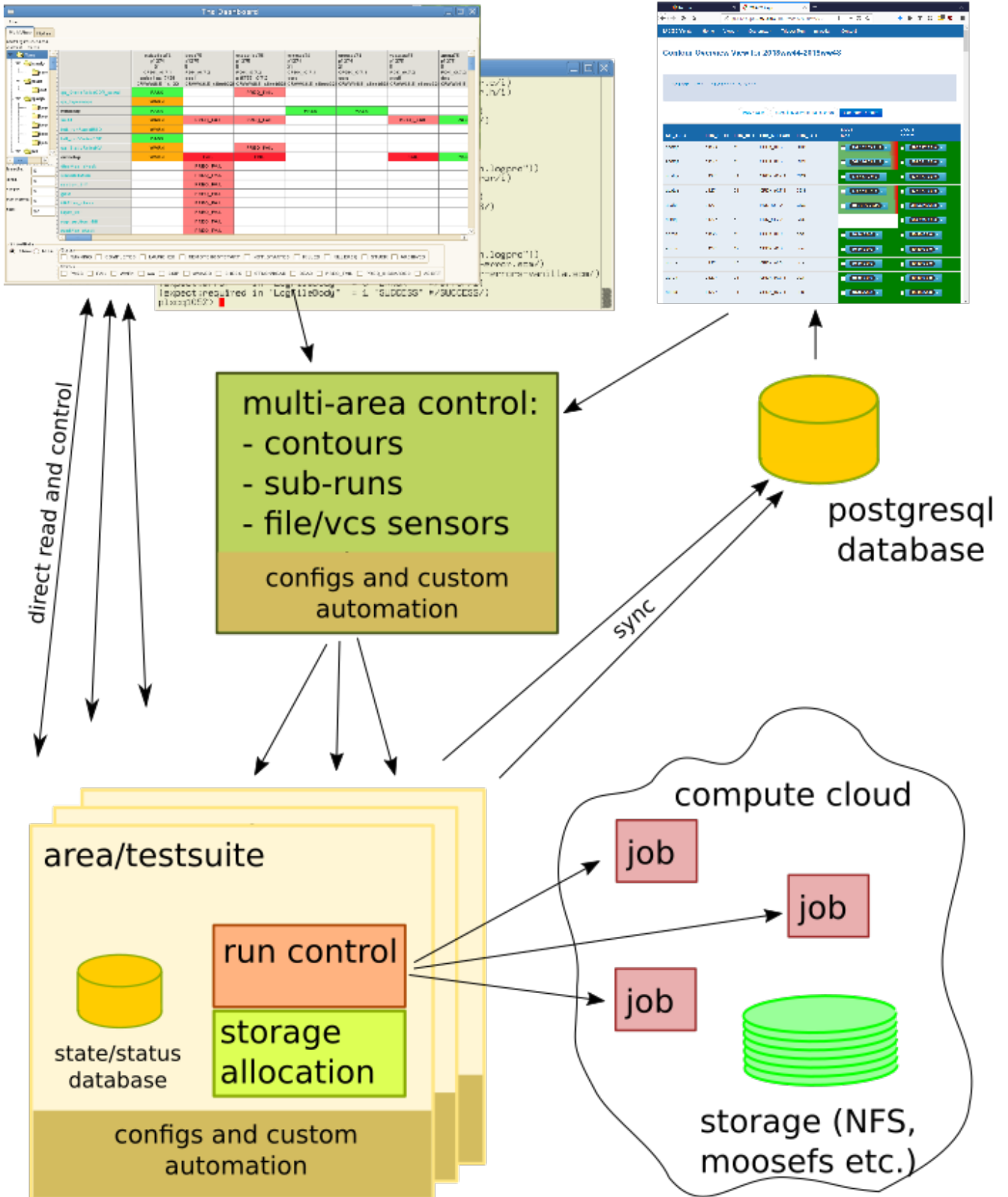
# Chapter 5

# TODO / Road Map

Note: This road-map is a wish list and not a formal plan. Items are in rough priority but are subject to change. Development is driven by user requests, developer "itch" and bug reports. Please contact matt@kiatoa.com with requests or bug reports. Requests from inside Intel generally take priority.

Dashboard and runs

1. Multi-area dashboard view

Tests Support

1. Add variable $MT_RUNPATH = $MT_LINKTREE/$MT_TARGET/$MT_RUNNAME

2. Improve [script], especially indent handling

Scalability

1. Overflow database methodology - combine the best of the v1.63 multi-db approach and the current db-in-tmp approach (currently slowness can be seen when number of tests in a db goes over 50-100k, with the overflow db it will be able to handle 1000's of runs with 50-100k tests per run). High priority - goal is to complete this by 20Q3.

Mtutils/CI

1. Enable mtutil calls from dashboard (for remote control)

2. Logs browser (esp. for surfacing mtutil related activities)

3. Embed ftfplan for distributed automation, completed activities trigger QA runs which trigger deployment etc.

4. Jenkins junit XML support [DONE]

5. Add output flushing in teamcity support

Build system

1. ./configure ⇒ ubuntu, sles11, sles12, rh7 [WIP]

2. Switch to using simple runs query everywhere

3. Add end_time to runs and add a rollup call that sets state, status and end_time

Code refactoring/quality/performance

1. Switch to scsh-process pipeline management for job execution/control

2. Use call-with-environment-variables where possible.

Migration to inmem db and or overflow db

1. Re-work the dbstruct data structure?

   a. [ run-id.db inmemdb last-mod last-read last-sync inuse ]

Some ideas for Megatest 2.0

1. Aggressive megatest.config and runconfig.config caching.

   a. Cache the configs in $MT_RUNPATH
   b. Following invocations of –run, -rerun* will calculate the new config but only overwrite the cached file IF changed

2. If the cached file changes ALL existing tests go from COMPLETED $\rightarrow$ STALE, I'm not sure what to do about RUNNING tests

3. !VARS in runconfigs are not exported to the environment. They are accessed via rget as if the ! was not there.

4. Per test copy commands (example is incomplete).

```
[testcopy]
%/iind% unison SRC DEST
% cp -r SRC DEST
```

Add ability to move runs to other Areas (overlaps with overflow db system)

1. allow shrinking megatest.db data by moving runs to an alternate Megatest area with same keys.

2. add param -destination [area|path]. when specified runs are copied to new area and removed from local db.

3. the data move would involve these steps

   a. copy the run data to destination area megatest.db
   b. mark the run records as deleted, do not remove the run data on disk

4. accessing the data would be by running dashboard in the satellite area

5. future versions of Megatest dashboard should support displaying areas in a merged way.

6. some new controls would be supported in the config

   a. [setup] $\Rightarrow$ allow-runs [no|yes] $\Longleftarrow$ used to disallow runs
   b. [setup] $\Rightarrow$ auto-migrate=[areaname|path] $\Longleftarrow$ used to automatically migrate data to a satellite area.

Eliminate ties to homehost (part of overflow db system)

1. Server creates captain pkt

2. Create a lock in the db

3. Relinquish db when done

Tasks - better management of run manager processes etc.

1. adjutant queries tasks table for next action [Migrate into mtutil]

    a. Task table used for tracking runner process [Replaced by mtutil]

    b. Task table used for jobs to run [Replaced by mtutil]

    c. Task table used for queueing runner actions (remove runs, cleanRunExecute, etc) [Replaced by mtutil]

2. adjutant (server/task dispatch/execution manager)

Stale propagation

1. Mark dependent tests for clean/rerun -rerun-downstream

2. On run start check for defunct tests in RUNNING, LAUNCHED or REMOTEHOSTSTART and correct or notify

3. Fix: refresh of gui sometimes fails on last item (race condition?)

Bin list

1. Rerun step and or subsequent steps from gui [DONE?]

2. Refresh test area files from gui

3. Clean and re-run button

4. Clean up STATE and STATUS handling.

    a. Dashboard and Test control panel are reverse order - choose and fix

    b. Move seldom used states and status to drop down selector

5. Access test control panel when clicking on Run Summary tests

6. Feature: -generate-index-tree

7. Change specifing of state and status to use STATE1/STATUS1,STATE2/STATUS2

8. rest api available for use with Perl, Ruby etc. scripts

9. megatest.config setup entries for:

    a. run launching (e.g. /bin/sh %CMD% > /dev/null)

    b. browser "konqueror %FNAME%

10. refdb: Add export of csv, json and sexp

11. Convert to using call-with-environment-variables where possible. Should allow handling of parallel runs in same process.

12. Re-work text interface wizards. Several bugs on record. Possibly convert to gui based.

13. Add to testconfig requirements section; launchlimiter scriptname, calls scriptname to check if ok to launch test

14. Refactor Run Summary view, currently very clumsy

15. Add option to show steps in Run Summary view

16. Refactor guis for resizeablity

17. Add filters to Run Summary view and Run Control view

18. Add to megatest.config or testconfig; rerunok STATE/STATUS,STATE/STATUS. . .

19. Launch gates for diskspace; /path/one>1G,/path/two>200M,/tmp>5G,#{scheme **toppath**}>1G

20. Tool tips

21. Filters on Run Summary, Summary and Run Control panel

22. Built in log viewer (partially implemented)

23. Refactor the test control panel Help and documentation

24. Complete the user manual (I've been working on this lately).

25. Online help in the gui Streamlined install

26. Deployed or static build

27. Added option to compile IUP (needed for VMs)

28. Server side run launching

29. Wizards for creating tests, regression areas (current ones are text only and limited).

30. Fully functional built in web service (currently you can browse runs but it is very simplistic).

31. Gui panels for editing megatest.config and runconfigs.config

32. Fully isolated tests (no use of NFS to see regression area files)

33. Windows version

# Chapter 6

# Installation

## 6.1 Dependencies

Chicken scheme and a number of "eggs" are required for building Megatest. See the script installall.sh in the utils directory of the source distribution for an automated way to install everything needed for building Megatest on Linux.

Megatest. In the v1.66 and beyond assistance to create the build system is built into the Makefile.

**Installation steps (overview)**

```
./configure
make chicken
setup.sh make -j install
```

Or install the needed build system manually:

1. Chicken scheme from http://call-cc.org

2. IUP from http://webserver2.tecgraf.puc-rio.br/iup/

3. CD from http://webserver2.tecgraf.puc-rio.br/cd/

4. IM from https://webserver2.tecgraf.puc-rio.br/im/

5. ffcall from http://webserver2.tecgraf.puc-rio.br/iup/

6. Nanomsg from https://nanomsg.org/ (NOTE: Plan is to eliminate nanomsg dependency).

7. Needed eggs (look at the eggs lists in the Makefile)

Then follow these steps:

**Installation steps (self-built chicken scheme build system)**

```
./configure
make -j install
```

# Chapter 7

# Getting Started

**Getting started with Megatest**

```
Creating a testsuite or flow and your first test or task.
```

After installing Megatest you can create a flow or testsuite and add some tests using the helpers. Here is a quickstart sequence to get you up and running your first automated testsuite.

## 7.1 Creating a Megatest Area

### 7.1.1 Choose Target Keys

First choose your "target" keys. These are used to organise your runs in a way that is meaningful to your project. If you are unsure about what to use for keys just use a single generic key such as "RUNTYPE". These keys will be used to hand values to your tests via environment variables so ensure they are unique. Prefixing them with something such as PROJKEYS_ is a good strategy.

Examples of keys:

Table 7.1: Example keys

| Option | Description |
|---|---|
| RELEASE/ITERATION | This example is used by Megatest for its internal QA. |
| ARCH/OS/RELEASE | For a software project targeting multiple platforms |
| UCTRLR/NODETYPE | Microcontroller project with different controllers running same software |

### 7.1.2 Create Area Config Files

You will need to choose locations for your runs (the data generated every time you run the testsuite) and link tree. For getting started answer the prompts with "runs" and "links". We use the Unix editor "vi" in the examples below but you can use any plain text editor.

**Using the helper to create a Megatest area**

```
megatest -create-megatest-area

# optional: verify that the settings are ok
vi megatest.config
vi runconfigs.config
```

## 7.2   Creating a Test

Choose the test name for your first test and run the helper. You can edit the files after the initial creation. You will need to enter names and scripts for the steps to be run and then edit the tests/<testname>/testconfig file and modify the logpro rules to properly process the log output from your steps. For your first test just hit enter for the "waiton", "priority" and iteration variable prompts.

Hint: for geting started make your logpro rules very liberal. expect:error patterns should match nothing and comment out expect:required rules.

**Using the helper to create a Megatest test**

```
megatest -create-test myfirsttest

# then edit the generated config
vi tests/myfirsttest/testconfig
```

## 7.3   Running your test

First choose a target and runname. If you have a two-place target such as RELEASE/ITERATION a target would look like v1.0/aff3 where v1.0 is the RELEASE and aff3 is the ITERATION. For a run name just use something like run1.

**Running all tests (testpatt of "%" matches all tests)**

```
megatest -run -target v1.0/aff3 -runname run1 -testpatt % -log run1.log
```

## 7.4   Viewing the results

Start the dashboard and browse your run in the "Runs" tab.

**Starting dashboard**

```
dashboard -rows 24
```

# Chapter 8

# Study Plan

Megatest is an extensive program with a lot to learn. Following are some paths through the material to smooth the learning path.

## 8.1 Basic Concepts (suggest you pick these up on the way)

- Components of automation; run, test, iteration
- Selectors; target, runname, and testpatt

## 8.2 Running Testsuites or Automation

- Using the dashboard gui (recommended)
  - Using the "Runs" panel.
  - Using the "Run Control" panel.
  - Using a test control panel
  - The Right Mouse Button menu
  - Debug features
    * xterm
      · pstree
      · log files; mt_copy.log, mt_launch.log
      · variables; megatest.csh, megatest.sh
      · testconfig dump, *testconfig
    * State/status buttons
    * Run, Clean, KillReq
    * ReRunClean
- Using the command line
  - Getting help; megatest -h, megatest -manual
  - Starting runs; megatest -run
    * Selection controls; -target, -runname and -testpatt

## 8.3   Writing Tests and Flows

- environment variables (table 5)

- tests/*testname*/testconfig testconfig details

  - ezsteps and logpro section
  - iteration (one test applied to many inputs), items, itemstable test iteration
  - dependencies, waiton, itemmatch, itemwait test requirements
  - miscellaneous; mode toplevel, runtimelim, skip on file, no file, script or on running, waiver propagation

- megatest areas

  - megatest.config
  - runconfigs.config
  - config language features; include, shell, system, scheme, rp|realpath, getenv, get, rget, scriptinc config file helpers

## 8.4   Advanced Topics

- Removing and keeping runs selectively managing runs

- Subruns nested runs

- Config file features config file features

- HTML output with -generate-html

- Triggers, post run, state/status

- MTLOWESTLOAD

- flexilauncher

- env delta and testconfig

- capturing test data, extracting values from logpro and using them for pass/fail

- mtutil, postgres connection, packets for cross-site/cross-user control (e.g. mcrun).

## 8.5   Maintenance and Troubleshooting

- cleanup-db, database structure of Megatest 1.6x

- archiving

- homehost management

- show-runconfig

- show-config

- show with -debug 0,9

- load management

# Chapter 9

# Writing Tests

## 9.1 Creating a new Test

The following steps will add a test "yourtestname" to your testsuite. This assumes starting from a directory where you already have a megatest.config and runconfigs.config.

1. Create a directory tests/yourtestname

2. Create a file tests/yourtestname/testconfig

**Contents of minimal testconfig**

```
[ezsteps]
stepname1 stepname.sh

# test_meta is a section for storing additional data on your test
[test_meta]
author myname
owner  myname
description An example test
reviewed never
```

This test runs a single step called "stepname1" which runs a script "stepname.sh". Note that although it is common to put the actions needed for a test step into a script it is not necessary.

# Chapter 10

# Debugging

## 10.1 Well Written Tests

### 10.1.1 Test Design and Surfacing Errors

Design your tests to surface errors. Ensure that all logs are processed by logpro (or a custom log processing tool) and can be reached by a mouse click or two from the test control panel.

To illustrate, here is a set of scripts with nested calls where script1.sh calls script2.sh which calls script3.sh which finally calls the Cadence EDA tool virtuoso:

**script1.sh**

```
#!/bin/bash
code ...
script2.sh some parameters > script2.log
more code ...
```

**script2.sh**

```
#!/bin/bash
code ...
script3.sh some more parameters > script3.log
more code ...
```

**script3.sh**

```
#!/bin/bash
code ...
virtuoso params and switches ...
more code ...
```

The log files script2.log, script3.log and the log output from virtuoso are not accessible from the test control panel. It would be much better for future users of your automation to use steps more fully. One easy option would be to post process the logs in downstream additional steps:

**testconfig**

```
[ezsteps]
step1 script1.sh
step2 cat script2.log
step3 cat script3.log
```

```
[logpro]
step1 ;; some logpro rules
   (expect:required in "LogFileBody" > 0 "Expect this output" #/something expected ↩
      /)
step2 ;; some logpro rules for script2.sh
step3 ;; some logpro rules for script3.sh

[scripts]
script1.sh #!/bin/bash
 code ...


...
```

With the above testconfig the logs for every critical part of the automation are fully surfaced and rules can be created to flag errors, warnings, aborts and to ignore false errors. A user of your automation will be able to see the important error with two mouse clicks from the runs view.

An even better would be to eliminate the nesting if possible. As a general statement with layers - less is usually more. By flattening the automation into a sequence of steps you can use the test control panel to re-run a step with a single click or from the test xterm run only the errant step from the command line.

The message here is make debugging and maintenace easy for future users (and yourself) by keeping clicks-to-error in mind.

## 10.2   Examining The Test Logs and Environment

### 10.2.1   Test Control Panel - xterm

From the dashboard click on a test PASS/FAIL button. This brings up a test control panel. Aproximately near the center left of the window there is a button "Start Xterm". Push this to get an xterm with the full context and environment loaded for that test. You can run scripts or ezsteps by copying from the testconfig (hint, load up the testconfig in a separate text editor window).

With more recent versions of Megatest you can step through your test from the test control panel. Click on the cell labeled "rerun this step" to only rerun the step or click on "restart from here" to rerun that step and downstream steps.

NOTE 1: visual feedback can take some time, give it a few seconds and you will see the step change color to blue as it starts running.

NOTE 2: steping through only works if you are using ezsteps.

## 10.3   A word on Bisecting

Bisecting is a debug strategy intended to speed up finding the root cause of some bug.
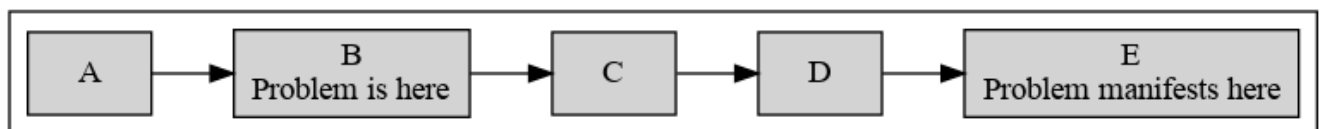


Figure 10.1: A complex process with a problem found in stage "E"

It is common to start debugging where the problem was observed and then work back. However by inspecting the output at stage "C" in the example above you would potentially save a lot of debug effort, this is similar to the feature in source control tools like git and fossil called biseceting.

## 10.4   Tough Bugs

Most bugs in Megatest based automation will be in the scripts called in your test steps and if you utilize the good design practice described above should be fairly easy for you to reproduce, isolate and find.

Some bugs however will come from subtle and hard to detect interactions between Megatest and your OS and Unix environment. This includes things like constructed variables that are legal in one context (e.g. tcsh) but illegal in another context (e.g. bash), variables that come from your login scripts and access and permissions issues (e.g. a script that silently fails due to no access to needed data). Other bugs might be due to Megatest itself.

To isolate bugs like this you may need to look at the log files at various stages in the execution process of your run and tests.
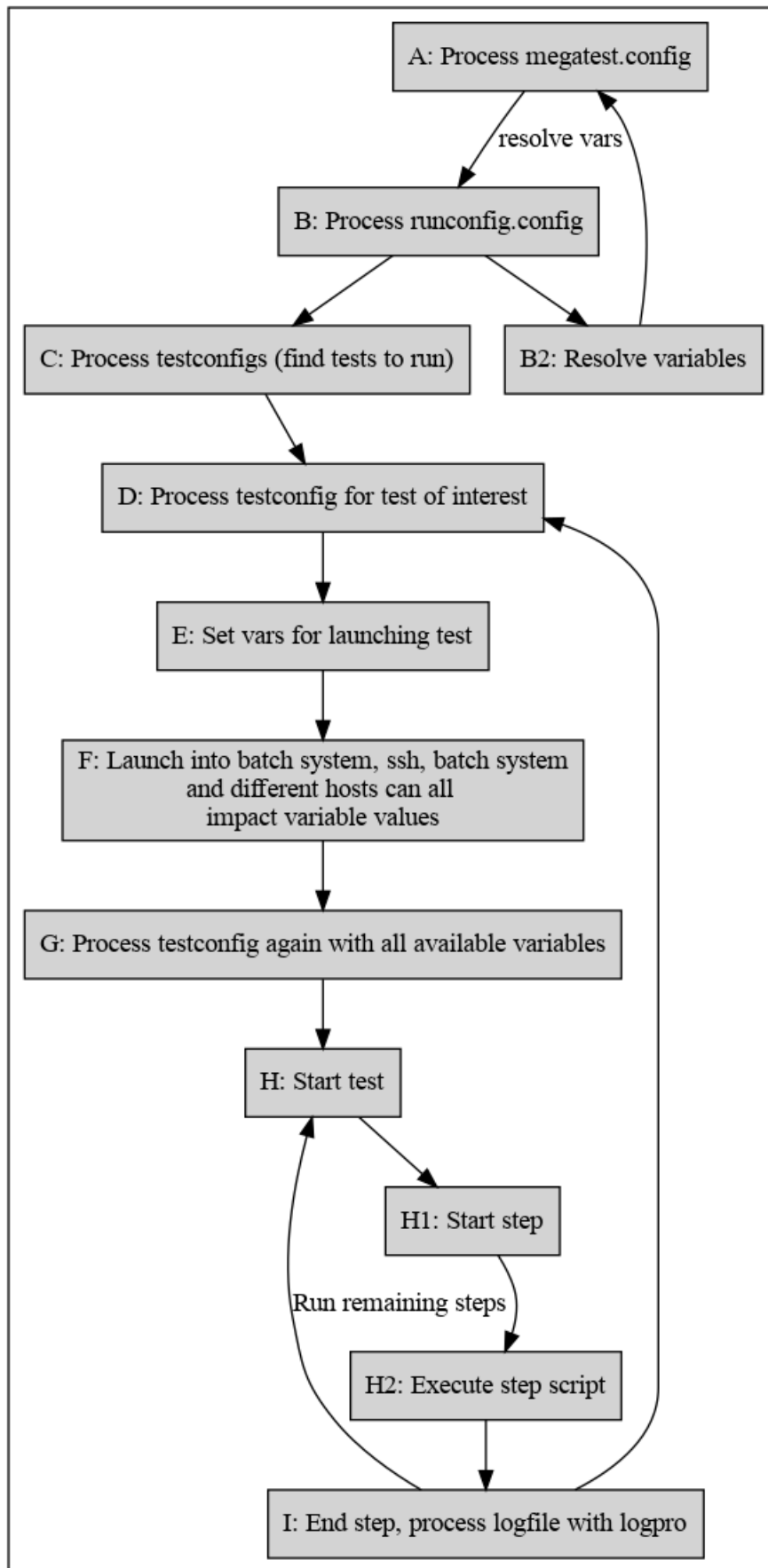
Figure 10.2: A simplified diagram of the stages Megatest goes through to run a test.

Table 10.1: How to check variable values and inspect logs at each stage

| Stage | How to inspect | Watch for or try ... |
|---|---|---|
| A: post config processing | `megatest -show-config -target your/target` | `#f (failed var processing)` |
| B: post runconfig | `megatest -show-runconfig -target your/target` | `Add -debug 0,9 to see which file your settings come from` |
| C: processing testconfigs | `inspect output from "megatest -run ..."` | `Messages indicating issues process configs, dependency problems` |
| D: process testconfig for test launch | `inspect output from megatest runner` | `Zero items (items expansion yielded no items)` |
| E,F: launching test | `start test xterm, look at mt_launch.log` | `Did your batch system accept the job?  Has the job landed on a machine?` |
| G: starting test | `look at your batch systems logs for the process` | `Did the megatest -execute process start and run?  Extract the "megatest -execute ..." command and run it from your xterm.` |
| H,H1,H2: step exectution | `look at <stepname>.log, <stepname>.html and your own internal logs` | `Do you have sufficiently tight logpro rules?  You must always have a "required" rule!` |

### 10.4.1  Bisecting megatest.csh/sh

Sometimes finding the environment variable that is causing the problem can be very difficult. Bisection can be applied.

Edit the megatest.csh or megatest.sh file and comment out 50% per round, source in fresh xterm and run the test.

This idea can also be applied to your .cshrc, .bashrc, .aliases and other similar files.

### 10.4.2  csh and -f

A common issue when tcsh or csh shells are used for scripting is to forget or choose to not use -f in your #! line.

**Not good**

```
#!/bin/tcsh
...
```

**Good**

```
#!/bin/tcsh -f
...
```

### 10.4.3  Config File Processing

As described above it is often helpful to know the content of variables in various contexts as Megatest works through the actions needed to run your tests. A handy technique is to force the startup of an xterm in the context being examined.

For example, if an item list is not being generated as expected you can inject the startup of an xterm as if it were an item:

**Original items table**

```
[items]
CELLNAME [system getcellname.sh]
```

**Items table modified for debug**

```
[items]
DEBUG [system xterm]
CELLNAME [system getcellnames.sh]
```

When this test is run an xterm will pop up. In that xterm the environment is exactly that in which the script "getcellnames.sh" would run. You can now debug the script to find out why it isn't working as expected.

Similarly in a script just call the xterm. NOTE: This technique can be very helpful in debugging running of EDA tools in Perl, Ruby, Python or tcl scripts:

**Perl example**

```
some_code();
$cmdline="virtuoso -some-switches and params ...";
print "$cmdline"; # print the command line so you can paste it into the xterm that ←
    pops up
system("xterm");  # this line is added for the debug and removed when done
system($cmdline);
more_code();
```

## 10.5  Misc Other Debugging Hints

### 10.5.1  Annotating scripts and config files

Run the "env" command to record the environment:

env | sort > stagename.log

In a config file:

**megatest.config, runconfigs.config and testconfig**

```
#{shell env | sort > stagename.log}

# or

[system env | sort > stagename.log]
```

In scripts just insert the commands, this example helps you identify if "some commands . . . " changed any environment variables.:

**myscript.sh**

```
env | sort > somefile-before.log
some commands ...
env | sort > somefile-after.log
```

**Use meld to examine the differences**

```
meld somefile-before.log somefile-after.log
```

### 10.5.2 Oneshot Modifying a Variable

To try various values for a variable without mutating the current value

**within a bash shell**

```
SOMEVAR=123 runcmd.sh
```

**within csh**

```
(setenv SOMEVAR 123;runcmd.sh)

# OR

env SOMEVAR=123 runcmd.sh
```

### 10.5.3 Capturing output from a command

**Use the "script" utility**

```
script -c "virtuoso -params and switches ..."
```

# Chapter 11

# How To Do Things

## 11.1 Process Runs

### 11.1.1 Remove Runs

From the dashboard click on the button (PASS/FAIL. . . )  for one of the tests. From the test control panel that comes up push the clean test button. The command field will be prefilled with a template command for removing that test. You can edit the command, for example change the argument to -testpatt to "%" to remove all tests.

**Remove the test diskperf and all it's items**

```
megatest -remove-runs -target ubuntu/nfs/none -runname ww28.1a -testpatt diskperf/% -v
```

**Remove all tests for all runs and all targets**

```
megatest -remove-runs -target %/%/% -runname % -testpatt % -v
```

### 11.1.2 Archive Runs

Megatest supports using the bup backup tool (https://bup.github.io/) to archive your tests for efficient storage and retrieval. Archived data can be rapidly retrieved if needed. The metadata for the run (PASS/FAIL status, run durations, time stamps etc.) are all preserved in the megatest database.

For setup information see the Archiving topic in the reference section of this manual.

#### 11.1.2.1 To Archive

Hint: use the test control panel to create a template command by pushing the "Archive Tests" button.

**Archive a full run**

```
megatest -target ubuntu/nfs/none -runname ww28.1a -archive save-remove -testpatt %
```

#### 11.1.2.2 To Restore

**Retrieve a single test**

```
megatest -target ubuntu/nfs/none -runname ww28.1a -archive restore -testpatt diskperf/%
```

Hint: You can browse the archive using bup commands directly.

```
bup -d /path/to/bup/archive ftp
```

## 11.2 Pass Data from Test to Test

**To save the data call archive save within your test:**

```
megatest -archive save
```

**To retrieve the data call archive get using patterns as needed**

```
# Put the retrieved data into /tmp
DESTPATH=/tmp/$USER/$MT_TARGET/$MT_RUN_NAME/$MT_TESTNAME/$MT_ITEMPATH/my_data
mkdir -p $DESTPATH
megatest -archive get -runname % -dest $DESTPATH
```

## 11.3 Submit jobs to Host Types based on Test Name

**In megatest.config**

```
[host-types]
general ssh #{getbgesthost general}
nbgeneral nbjob run JOBCOMMAND -log $MT_LINKTREE/$MT_TARGET/$MT_RUNNAME.$MT_TESTNAME- ←
    $MT_ITEM_PATH.lgo

[hosts]
general cubian xena

[launchers]
envsetup general
xor/%/n 4C16G
% nbgeneral

[jobtools]
launcher bsub
# if defined and not "no" flexi-launcher will bypass launcher unless there is no
# match.
flexi-launcher yes
```

# Chapter 12

# Tricks and Tips

This section is a collection of a various useful tricks for that didn't quite fit elsewhere.

## 12.1  Limiting your running jobs

The following example will limit a test in the jobgroup "group1" to no more than 10 tests simultaneously.

In your testconfig:

```
[test_meta]
jobgroup group1
```

In your megatest.config:

```
[jobgroups]
group1 10
custdes 4
```

### 12.1.1  Organising Your Tests and Tasks

The default location "tests" for storing tests can be extended by adding to your tests-paths section.

```
[misc]
parent #{shell dirname $(readlink -f .)}

[tests-paths]
1 #{get misc parent}/simplerun/tests
```

The above example shows how you can use addition sections in your config file to do complex processing. By putting results of relatively slow operations into variables the processing of your configs can be kept fast.

### 12.1.2  Alternative Method for Running your Job Script

**Directly running job in testconfig**

```
[setup]
runscript main.csh
```

The runscript method is essentially a brute force way to run scripts where the user is responsible for setting STATE and STATUS and managing the details of running a test.

## 12.2 Debugging Server Problems

Some handy Unix commands to track down issues with servers not communicating with your test manager processes. Please put in tickets at https://www.kiatoa.com/fossils/megatest if you have problems with servers getting stuck.

```
sudo lsof -i
sudo netstat -lptu
sudo netstat -tulpn
```

# Chapter 13

# Reference

## 13.1   Megatest Use Modes

Table 13.1: Base commands

| Use case | Megatest command | mtutil |
|---|---|---|
| Start from scratch | `-rerun-all` | `restart` |
| Rerun non-good completed | `-rerun-clean` | `rerunclean` |
| Rerun all non-good and not completed yet | `-set-state-status KILLREQ; -rerun-` | `clean` |
| killrerun | `Continue run` | `-run` |
| resume | `Remove run` | `-remove-runs` |
| clean | `Lock run` | `-lock` |
| lock | `Unlock run` | `-unlock` |
| unlock | `killrun` | `-set-state-status KILLREQ; -kill-run` |

## 13.2   Config File Helpers

Various helpers for more advanced config files.

Table 13.2: Helpers

| Helper | Purpose | Valid values | Comments |
|---|---|---|---|
| #{scheme (scheme code...)} | `Execute arbitrary scheme code` | `Any valid scheme` | `Value returned from the call is converted to a string and processed as part of the config file` |

Table 13.2: (continued)

| Helper | Purpose | Valid values | Comments |
|---|---|---|---|
| #{system command} | Execute program, inserts exit code | Any valid Unix command | Discards the output from the program |
| #{shell command} or #{sh ...} | Execute program, inserts result from stdout | Any valid Unix command | Value returned from the call is converted to a string and processed as part of the config file |
| #{realpath path} or #{rp ...} | Replace with normalized path | Must be a valid path | |
| #{getenv VAR} or #{gv VAR} | Replace with content of env variable | Must be a valid var | |
| #{get s v} or #{g s v} | Replace with variable v from section s | Variable must be defined before use | |
| #{rget v} | Replace with variable v from target or default of runconfigs file | | |
| | Replace with the path to the megatest testsuite area | | |

## 13.3 Config File Settings

Settings in megatest.config

## 13.4 Config File Additional Features

Including output from a script as if it was inline to the config file:

```
[scriptinc myscript.sh]
```

If the script outputs:

```
[items]
A a b c
B d e f
```

Then the config file would effectively appear to contain an items section exactly like the output from the script. This is useful when dynamically creating items, itemstables and other config structures. You can see the expansion of the call by looking in the cached files (look in your linktree for megatest.config and runconfigs.config cache files and in your test run areas for the expanded and cached testconfig).

Wildcards and regexes in Targets

```
[a/2/b]
VAR1 VAL1

[a/%/b]
VAR1 VAL2
```

Will result in:

```
[a/2/b]
VAR1 VAL2
```

Can use either wildcard of "%" or a regular expression:

```
[/abc.*def/]
```

## 13.5 Disk Space Checks

Some parameters you can put in the [setup] section of megatest.config:

```
# minimum space required in a run disk
minspace 10000000

# minimum space required in dbdir:
dbdir-space-required 100000

# script that takes path as parameter and returns number of bytes available:
free-space-script check-space.sh
```

## 13.6 Trim trailing spaces

**Note**

As of Megatest version v1.6548 trim-trailing-spaces defaults to yes.

```
[configf:settings trim-trailing-spaces no]
#                 |<== next line padded with spaces to here
DEFAULT_INDENT
[configf:settings trim-trailing-spaces no]
```

The variable DEFAULT_INDENT would be a string of 3 spaces

## 13.7 Job Submission Control

### 13.7.1 Submit jobs to Host Types based on Test Name

**In megatest.config**

```
[host-types]
general   nbfake
remote    bsub

[launchers]
```

```
runfirst/sum% remote
% general

[jobtools]
launcher bsub
# if defined and not "no" flexi-launcher will bypass launcher unless
# there is no host-type match.
flexi-launcher yes
```

#### 13.7.1.1 host-types

List of host types and the commandline to run a job on that host type.

**host-type ⇒ launch command**

```
general nbfake
```

#### 13.7.1.2 launchers

**test/itempath ⇒ host-type**

```
runfirst/sum% remote
```

#### 13.7.1.3 Miscellaneous Setup Items

Attempt to rerun tests in "STUCK/DEAD", "n/a", "ZERO_ITEMS" states.

**In megatest.config**

```
[setup]
reruns 5
```

Replace the default blacklisted environment variables with user supplied list.

Default list: USER HOME DISPLAY LS_COLORS XKEYSYMDB EDITOR MAKEFLAGS MAKEF MAKEOVERRIDES

**Add a "bad" variable "PROMPT" to the variables that will be commented out** in the megatest.sh and megatest.csh files:

```
[setup]
blacklistvars USER HOME DISPLAY LS_COLORS XKEYSYMDB EDITOR MAKEFLAGS PROMPT
```

#### 13.7.1.4 Run time limit

```
[setup]
# this will automatically kill the test if it runs for more than 1h 2m and 3s
runtimelim 1h 2m 3s
```

#### 13.7.1.5 Post Run Hook

This runs script to-run.sh after all tests have been completed. It is not necessary to use -run-wait as each test will check for other running tests on completion and if there are none it will call the post run hook.

Note that the output from the script call will be placed in a log file in the logs directory with a file name derived by replacing / with _ in post-hook-<target>-<runname>.log.

```
[runs]
post-hook /path/to/script/to-run.sh
```

## 13.8 Tests browser view

The tests browser (see the Run Control tab on the dashboard) has two views for displaying the tests.

1. Dot (graphviz) based tree

2. No dot, plain listing

The default is the graphviz based tree but if your tests don't view well in that mode then use "nodot" to turn it off.

```
[setup]
nodot
```

## 13.9 Capturing Test Data

In a test you can capture arbitrary variables and roll them up in the megatest database for viewing on the dashboard or web app.

**In a test as a script**

```
$MT_MEGATEST -load-test-data << EOF
foo,bar,   1.2,  1.9, >
foo,rab, 1.0e9, 10e9, 1e9
foo,bla,   1.2,  1.9, <
foo,bal,   1.2,  1.2, <   ,      ,Check for overload
foo,alb,   1.2,  1.2, <=  , Amps,This is the high power circuit test
foo,abl,   1.2,  1.3, 0.1
foo,bra,   1.2, pass, silly stuff
faz,bar,    10,  8mA,      ,      ,"this is a comment"
EOF
```

Alternatively you can use logpro triggers to capture values and inject them into megatest using the -set-values mechanism:

**Megatest help related to -set-values**

```
Test data capture
  -set-values              : update or set values in the testdata table
  :category                : set the category field (optional)
  :variable                : set the variable name (optional)
  :value                   : value measured (required)
  :expected                : value expected (required)
  :tol                     : |value-expect| <= tol (required, can be <, >, >=, <= or number)
  :units                   : name of the units for value, expected_value etc. (optional)
```

## 13.10 Dashboard settings

**Runs tab buttons, font and size**

```
[dashboard]
btn-height x14
btn-fontsz 10
cell-width 60
```

## 13.11  Database settings

Table 13.3: Database config settings in [setup] section of megatest.config

| Var | Purpose | Valid values | Comments |
|-----|---------|--------------|----------|
| delay-on-busy | Prevent concurrent access issues | yes\|no or not defined | Default=no, may help on some network file systems, may slow things down also. |
| faststart | All direct file access to sqlite db files | yes\|no or not defined | Default=yes, suggest no for central automated systems and yes for interactive use |
| homehost | Start servers on this host | <hostname> | Defaults to local host |
| hostname | Hostname to bind to | <hostname>\|- | On multi-homed hosts allows binding to specific hostname |
| lowport | Start searching for a port at this portnum | 32768 | |
| required | Server required | yes\|no or not defined | Default=no, force start of server always |
| server-query-threshold | Start server when queries take longer than this | number in milliseconds | Default=300 |
| timeout | http api timeout | number in hours | Default is 1 minute, do not change |

# Chapter 14

# The testconfig File

## 14.1 Setup section

### 14.1.1 Header

```
[setup]
```

The runscript method is a brute force way to run scripts where the user is responsible for setting STATE and STATUS

```
runscript main.csh
```

## 14.2 Iteration

**Sections for iteration**

```
# full combinations
[items]
A x y
B 1 2

# Yields: x/1 x/2 y/1 y/2

# tabled
[itemstable]
A x y
B 1 2

# Yields x/1 y/2
```

**Or use files**

```
[itemopts]
slash path/to/file/with/items
# or
space path/to/file/with/items
```

**File format for / delimited**

```
key1/key2/key3
val1/val2/val2
...
```

**File format for space delimited**

```
key1 key2 key3
val1 val2 val2
...
```

## 14.3  Requirements section

**Header**

```
[requirements]
```

## 14.4  Wait on Other Tests

```
# A normal waiton waits for the prior tests to be COMPLETED
# and PASS, CHECK or WAIVED
waiton test1 test2
```

---

**Note**
Dynamic waiton lists must be capable of being calculated at the beginning of a run. This is because Megatest walks the tree of waitons to create the list of tests to execute.

---

**This works**

```
waiton [system somescript.sh]
```

**This does NOT work (the full context for the test is not available so #{shell . . . } is NOT enabled to evaluate.**

```
waiton #{shell somescript.sh}
```

**This does NOT work**

```
waiton [system somescript_that_depends_on_a_prior_test.sh]
```

## 14.5  Mode

The default (i.e. if mode is not specified) is normal. All pre-dependent tests must be COMPLETED and PASS, CHECK or WAIVED before the test will start

```
[requirements]
mode    normal
```

The toplevel mode requires only that the prior tests are COMPLETED.

```
[requirements]
mode toplevel
```

A item based waiton will start items in a test when the same-named item is COMPLETED and PASS, CHECK or WAIVED in the prior test. This was historically called "itemwait" mode. The terms "itemwait" and "itemmatch" are synonyms.

```
[requirements]
mode itemmatch
```

## 14.6 Overriding Enviroment Variables

Override variables before starting the test. Can include files (perhaps generated by megatest -envdelta or similar).

```
[pre-launch-env-vars]
VAR1 value1

# Get some generated settings
[include ../generated-vars.config]

# Use this trick to unset variables
#{scheme (unsetenv "FOOBAR")}
```

## 14.7 Itemmap Handling

For cases were the dependent test has a similar but not identical itempath to the downstream test an itemmap can allow for itemmatch mode

**example for removing part of itemmap for waiton test (eg: item foo-x/bar depends on waiton's item y/bar)**

```
[requirements]
mode itemwait
# itemmap <item pattern for this test>  <item replacement pattern for waiton test>
itemmap .*x/ y/
```

**example for removing part of itemmap for waiton test (eg: item foo/bar/baz in this test depends on waiton's item baz)**

```
# ## pattern replacement notes
#
# ## Example
# ## Remove everything up to the last /
[requirements]
mode itemwait
# itemmap <item pattern for this test> <nothing here indicates removal>
itemmap .*/
```

**example replacing part of itemmap for (eg: item foo/1234 will imply waiton's item bar/1234)**

```
#
# ## Example
# ## Replace foo/ with bar/
[requirements]
mode itemwait
# itemmap <item pattern for this test>  <item replacement pattern for waiton test>
itemmap foo/ bar/
```

**example for backreference (eg: item foo23/thud will imply waiton's item num-23/bar/thud**

```
#
# ## Example
# ## can use {number} in replacement pattern to backreference a (capture) from matching  ←
   pattern similar to sed or perl
[requirements]
mode itemwait
# itemmap <item pattern for this test>  <item replacement pattern for waiton test>
itemmap foo(\d+)/ num-\1/bar/
```

**example multiple itemmaps**

```
# multi-line; matches are applied in the listed order
# The following would map:
#   a123b321 to b321fooa123 then to 321fooa123p
#
[requirements]
itemmap (a\d+)(b\d+) \2foo\1
  b(.*) \1p
```

## 14.8  Complex mapping

Complex mappings can be handled with a separate [itemmap] section (instead if an itemmap line in the [requirements] section)

Each line in an itemmap section starts with a waiton test name followed by an itemmap expression

**eg: The following causes waiton test A item bar/1234 to run when our test's foo/1234 item is requested as well as causing waiton test B's blah item to run when our test's stuff/blah item is requested**

```
[itemmap]
A foo/ bar/
B stuff/
```

## 14.9 Complex mapping example



Complex Itemmapping (arrows indicate order of execution)
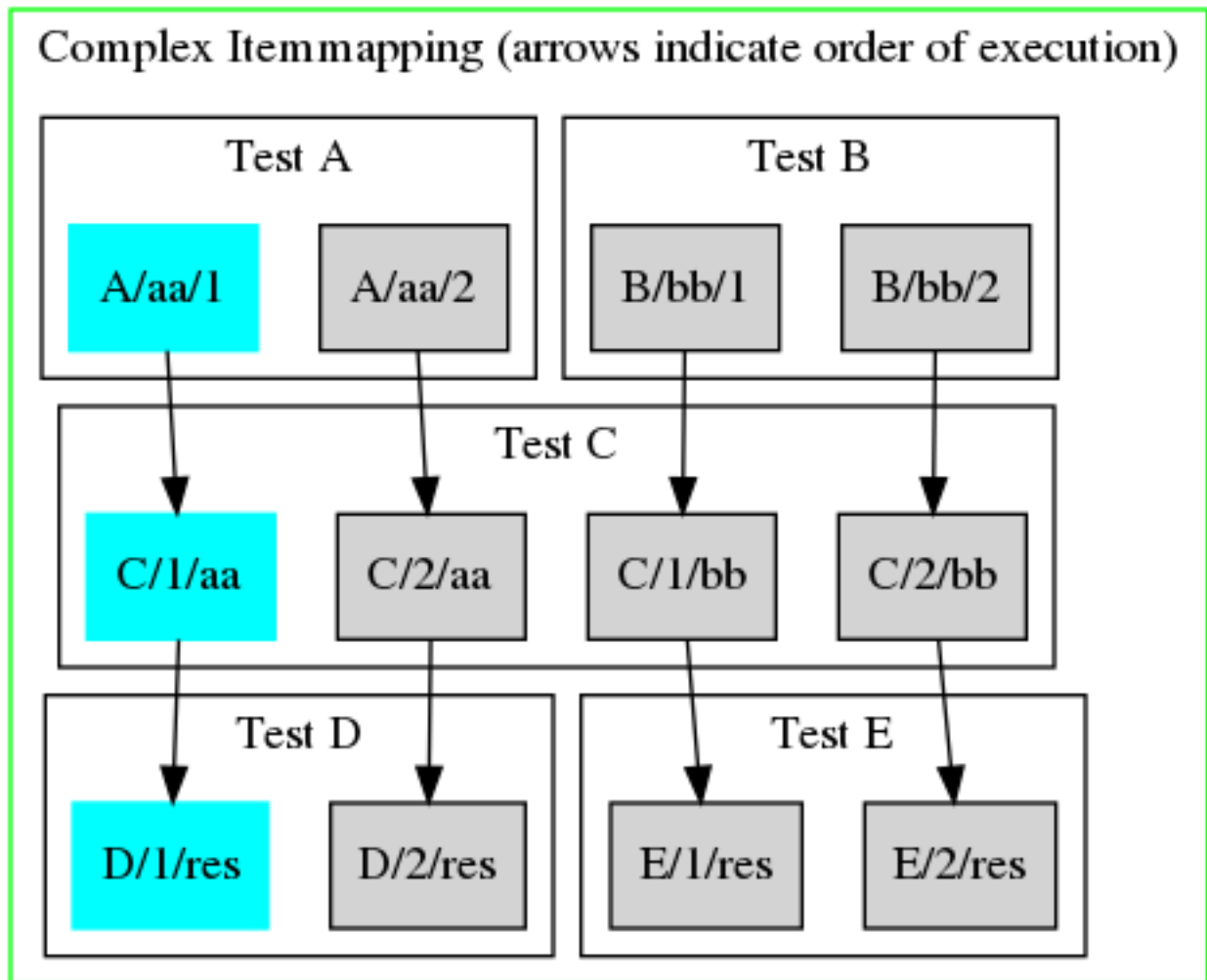
We accomplish this by configuring the testconfigs of our tests C D and E as follows:

**Testconfig for Test E has**

```
[requirements]
waiton C
itemmap (\d+)/res \1/bb
```

**Testconfig for Test D has**

```
[requirements]
waiton C
itemmap (\d+)/res \1/aa
```

**Testconfig for Test C has**

```
[requirements]
waiton A B

[itemmap]
A (\d+)/aa aa/\1
B (\d+)/bb bb/\1
```

**Testconfigs for Test B and Test A have no waiton or itemmap configured**

WALK THROUGH ONE ITEM — WE WANT THE FOLLOWING TO HAPPEN FOR TESTPATT `D/1/RES` (SEE BLUE BOXES IN COMPLEX ITEMMAPING FIGURE ABOVE):

1. eg from command line `megatest -run -testpatt D/1/res -target mytarget -runname myrunname`

2. Full list to be run is now: `D/1/res`

3. Test D has a waiton - test C. Test D's itemmap rule `itemmap (\d+)/res \1/aa` → causes `C/1/aa` to run before `D/1/res`

4. Full list to be run is now: `D/1/res, C/1/aa`

5. Test C was a waiton - test A. Test C's rule `A (\d+)/aa aa/\1` → causes `A/aa/1` to run before `C/1/aa`

6. Full list to be run is now: `D/1/res, C/1/aa, A/aa/1`

7. Test A has no waitons. All waitons of all tests in full list have been processed. Full list is finalized.

## 14.10   itemstable

An alternative to defining items is the itemstable section. This lets you define the itempath in a table format rather than specifying components and relying on getting all permutations of those components.

## 14.11   Dynamic Flow Dependency Tree

**Autogeneration waiton list for dynamic flow dependency trees**

```
[requirements]
# With a toplevel test you may wish to generate your list
# of tests to run dynamically
#
waiton #{shell get-valid-tests-to-run.sh}
```

## 14.12   Run time limit

```
[requirements]
runtimelim 1h 2m 3s  # this will automatically kill the test if it runs for more than 1h 2m ↩
    and 3s
```

## 14.13   Skip

A test with a skip section will conditional skip running.

**Skip section example**

```
[skip]
prevrunning x
# rundelay 30m 15s
```

## 14.14   Skip on Still-running Tests

```
# NB// If the prevrunning line exists with *any* value the test will
# automatically SKIP if the same-named test is currently RUNNING. The
# "x" can be any string. Comment out the prevrunning line to turn off
# skip.

[skip]
prevrunning x
```

## 14.15   Skip if a File Exists

```
[skip]
fileexists /path/to/a/file # skip if /path/to/a/file exists
```

## 14.16   Skip if a File Does not Exist

```
[skip]
filenotexists /path/to/a/file # skip if /path/to/a/file does not exist
```

## 14.17   Skip if a script completes with 0 status

```
[skip]
script /path/to/a/script # skip if /path/to/a/script completes with 0 status
```

## 14.18   Skip if test ran more recently than specified time

**Skip if this test has been run in the past fifteen minutes and 15 seconds.**

```
[skip]
rundelay 15m 15s
```

## 14.19   Disks

A disks section in testconfig will override the disks section in megatest.config. This can be used to allocate disks on a per-test or per item basis.

## 14.20   Controlled waiver propagation

If test is FAIL and previous test in run with same MT_TARGET is WAIVED or if the test/itempath is listed under the matching target in the waivers roll forward file (see below for file spec) then apply the following rules from the testconfig: If a waiver check is specified in the testconfig apply the check and if it passes then set this FAIL to WAIVED

Waiver check has two parts, 1) a list of waiver, rulename, filepatterns and 2) the rulename script spec (note that "diff" and "logpro" are predefined)

```
###### EXAMPLE FROM testconfig #########
# matching file(s) will be diff'd with previous run and logpro applied
# if PASS or WARN result from logpro then WAIVER state is set
#
[waivers]
# logpro_file    rulename       input_glob
waiver_1         logpro         lookittmp.log

[waiver_rules]

# This builtin rule is the default if there is no <waivername>.logpro file
# diff   diff %file1% %file2%

# This builtin rule is applied if a <waivername>.logpro file exists
# logpro diff %file1% %file2% | logpro %waivername%.logpro %waivername%.html
```

### 14.20.1 Waiver roll-forward files

To transfer waivers from one Megatest area to another it is possible to dump waivers into a file and reference that file in another area.

**Dumping the waivers**

```
megatest -list-waivers -runname %-a > mywaivers.dat
```

**Referencing the saved waivers**

```
# In megatest.config, all files listed will be loaded - recomended to use
# variables to select directorys to minimize what gets loaded.
[setup]
waivers-dirs /path/to/waiver/files /another/path/to/waiver/files
```

**Waiver files format**

```
[the/target/here]
# comments are fine
testname1/itempath A comment about why it was waived
testname2          A comment for a non-itemized test
```

## 14.21 Ezsteps

**Example ezsteps with logpro rules**

```
[ezsteps]
lookittmp   ls /tmp

[logpro]
lookittmp ;; Note: config file format supports multi-line entries where leading whitespace  ←
   is removed from each line
  ;;      a blank line indicates the end of the block of text
  (expect:required in "LogFileBody" > 0 "A file name that should never exist!" #/This is a  ←
     awfully stupid file name that should never be found in the temp dir/)
```

To transfer the environment to the next step you can do the following: # if your upstream file is csh you can force csh like this # if your upstream is bash loadenv source $REF/ourenviron.sh

**Propagate environment to next step**

```
$MT_MEGATEST -env2file .ezsteps/${stepname}
```

## 14.22 Scripts

**Specifying scripts inline (best used for only simple scripts)**

```
[scripts]
loaddb #!/bin/bash
  sqlite3 $1 <<EOF
  .mode tabs
  .import $2 data
  .q
  EOF
```

The above snippet results in the creation of an executable script called "loaddb" in the test directory. NOTE: every line in the script must be prefixed with the exact same number of spaces. Lines beginning with a # will not work as expected. Currently you cannot indent intermediate lines.

**Full example with ezsteps, logpro rules, scripts etc.**

```
# You can include a common file
#
[include #{getenv MT_RUN_AREA_HOME}/global-testconfig.inc]

# Use "var" for a scratch pad
#
[var]
dumpsql select * from data;
sepstr ...................................

# NOT IMPLEMENTED YET!
#
[ezsteps-addendum]
prescript something.sh
postscript something2.sh

# Add additional steps here. Format is "stepname script"
[ezsteps]
importdb loaddb prod.db prod.sql
dumpprod dumpdata prod.db "#{get var dumpsql}"
diff (echo "prod#{get var sepstr}test";diff --side-by-side \
     dumpprod.log reference.log ;echo DIFFDONE)

[scripts]
loaddb #!/bin/bash
  sqlite3 $1 <<EOF
  .mode tabs
  .import $2 data
  .q
  EOF

dumpdata #!/bin/bash
  sqlite3 $1 <<EOF
  .separator ,
  $2
  .q
  EOF

# Test requirements are specified here
[requirements]
waiton setup
priority 0

# Iteration for your test is controlled by the items section
```

```
# The complicated if is needed to allow processing of the config for the dashboard when  ↩
    there are no actual runs.
[items]
THINGNAME [system generatethings.sh | sort -u]

# Logpro rules for each step can be captured here in the testconfig
# note: The ;; after the stepname and the leading whitespace are required
#
[logpro]
inputdb ;;
  (expect:ignore   in "LogFileBody"  < 99 "Ignore error in comments"       #/^\/\/.*error/)
  (expect:warning  in "LogFileBody"  = 0 "Any warning"                     #/warn/)
  (expect:required in "LogFileBody"  > 0 "Some data found"                 #/^[a-z ↩
     ]{3,4}[0-9]+_r.*/)

diff ;;
  (expect:ignore   in "LogFileBody"  < 99 "Ignore error in comments"       #/^\/\/.*error/)
  (expect:warning  in "LogFileBody"  = 0 "Any warning"                     #/warn/)
  (expect:error    in "LogFileBody"  = 0 "< or > indicate missing entry"  (list #/(<|>)/    ↩
     #/error/i))
  (expect:error    in "LogFileBody"  = 0 "Difference in data"             (list #/\s+\|\s+/ ↩
       #/error/i))
  (expect:required in "LogFileBody"  > 0 "DIFFDONE Marker found"           #/DIFFDONE/)
  (expect:required in "LogFileBody"  > 0 "Some things found"               #/^[a-z ↩
     ]{3,4}[0-9]+_r.*/)

# NOT IMPLEMENTED YET!
#
## Also: enhance logpro to take list of command files: file1,file2...
[waivers]
createprod{target=%78/%/%/%} ;;
  (disable:required "DIFFDONE Marker found")
  (disable:error    "Some error")
  (expect:waive  in "LogFileBody" < 99 "Waive if failed due to version" #/\w+3\.6.*/)

# test_meta is a section for storing additional data on your test
[test_meta]
author matt
owner  matt
description Compare things
tags tagone,tagtwo
reviewed never
```

## 14.23  Triggers

In your testconfig or megatest.config triggers can be specified

**Triggers spec**

```
[triggers]

# Call script running.sh when test goes to state=RUNNING, status=PASS
RUNNING/PASS running.sh

# Call script running.sh any time state goes to RUNNING
RUNNING/ running.sh

# Call script onpass.sh any time status goes to PASS
PASS/ onpass.sh
```

Scripts called will have; test-id test-rundir trigger test-name item-path state status event-time, added to the commandline.

HINT

To start an xterm (useful for debugging), use a command line like the following:

**Start an xterm using a trigger for test completed.**

```
[triggers]
COMPLETED/ xterm -e bash -s --
```

---

**Note**
There is a trailing space after the double-dash

---

There are a number of environment variables available to the trigger script but since triggers can be called in various contexts not all variables are available at all times. The trigger script should check for the variable and fail gracefully if it doesn't exist.

Table 14.1: Environment variables visible to the trigger script

| Variable | Purpose |
| --- | --- |
| MT_TEST_RUN_DIR | The directory where Megatest ran this test |
| MT_CMDINFO | Encoded command data for the test |
| MT_DEBUG_MODE | Used to pass the debug mode to nested calls to Megatest |
| MT_RUN_AREA_HOME | Megatest home area |
| MT_TESTSUITENAME | The name of this testsuite or area |
| MT_TEST_NAME | The name of this test |
| MT_ITEM_INFO | The variable and values for the test item |
| MT_MEGATEST | Which Megatest binary is being used by this area |
| MT_TARGET | The target variable values, separated by / |
| MT_LINKTREE | The base of the link tree where all run tests can be found |
| MT_ITEMPATH | The values of the item path variables, separated by / |
| MT_RUNNAME | The name of the run |

## 14.24 Override the Toplevel HTML File

Megatest generates a simple html file summary for top level tests of iterated tests. The generation can be overridden. NOTE: the output of the script is captured from stdout to create the html.

**For test "runfirst" override the toplevel generation with a script "mysummary.sh"**

```
# Override the rollup for specific tests
[testrollup]
runfirst mysummary.sh
```

```
[triggers]
```

# Chapter 15

# Archiving Setup

In megatest.config add the following sections:

**megatest.config**

```
[archive]
# where to get bup executable
# bup /path/to/bup

[archive-disks]

# Archives will be organised under these paths like this:
#   <testsuite>/<creationdate>
# Within the archive the data is structured like this:
#   <target>/<runname>/<test>/
archive0 /mfs/myarchive-data/adisk1
```

# Chapter 16

# Environment Variables

It is often necessary to capture and or manipulate environment variables. Megatest has some facilities built in to help.

## 16.1 Capture variables

**Commands**

```
# capture the current enviroment into a db called envdat.db under
# the context "before"
megatest -envcap before

# capture the current environment into a db called startup.db with
# context "after"
megatest -envcap after startup.db

# write the diff from before to after
megatest -envdelta before-after -dumpmode bash
```

Dump modes include bash, csh and config. You can include config data into megatest.config, runconfigs.config and testconfig files. This is useful for capturing a complex environment in a special-purpose test and then utilizing that environment in downstream tests.

**Example of generating and using config data**

```
megatest -envcap original
# do some stuff here
megatest -envcap munged
megatest -envdelta original-munged -dumpmode ini -o modified.config
```

Then in runconfigs.config

**Example of using modified.config in a testconfig**

```
[pre-launch-env-vars]
[include modified.config]
```

# Chapter 17

# Managing Old Runs

It is often desired to keep some older runs around but this must be balanced with the costs of disk space.

1. Use -remove-keep

2. Use -archive (can also be done from the -remove-keep interface)

3. use -remove-runs with -keep-records

**For each target, remove all runs but the most recent 3 if they are over 1 week old**

```
# use –precmd 'sleep 5;nbfake' to limit overloading the host computer but to allow the  ←
    removes to run in parallel.
megatest –actions print,remove-runs –remove-keep 3 –target %/%/%/% –runname % –age 1w – ←
    precmd 'sleep 5;nbfake'"
```

# Chapter 18

# Nested Runs

A Megatest test can run a full Megatest run in either the same Megatest area or in another area. This is a powerful way of chaining complex suites of tests and or actions.

If you are not using the current area you can use ezsteps to retrieve and setup the sub-Megatest run area.

In the testconfig:

```
[subrun]

# Required: wait for the run or just launch it
#           if no then the run will be an automatic PASS irrespective of the actual result
run-wait yes|no

# Optional: where to execute the run. Default is the current runarea
run-area /some/path/to/megatest/area

# Optional: method to use to determine pass/fail status of the run
#   auto (default) - roll up the net state/status of the sub-run
#   logpro         - use the provided logpro rules, happens automatically if there is a  ←
    logpro section
# passfail auto|logpro
# Example of logpro:
passfail logpro

# Optional:
logpro ;; if this section exists then logpro is used to determine pass/fail
  (expect:required in "LogFileBody" >= 1 "At least one pass" #/PASS/)
  (expect:error     in "LogFileBody"  = 0 "No FAILs allowed"  #/FAIL/)

# Optional: target translator, default is to use the parent target
target #{shell somescript.sh}

# Optional: runname translator/generator, default is to use the parent runname
run-name #{somescript.sh}

# Optional: testpatt spec, default is to first look for TESTPATT spec from runconfigs  ←
    unless there is a contour spec
test-patt %/item1,test2

# Optional: contour spec, use the named contour from the megatest.config contour spec
contour contourname ### NOTE: Not implemented yet! Let us know if you need this feature.

# Optional: mode-patt, use this spec for testpatt from runconfigs
mode-patt TESTPATT
```

```
# Optional: tag-expr, use this tag-expr to select tests
tag-expr quick

# Optional: (not yet implemented, remove-runs is always propagated at this time), propagate ←
    these actions from the parent
#          test
#   Note// default is % for all
propagate remove-runs archive ...
```

# Chapter 19

# Programming API

These routines can be called from the megatest repl.

Table 19.1: API Keys Related Calls

| API Call | Purpose comments | Returns | Comments |
|---|---|---|---|
| (rmt:get-<br>keys<br>run-id) | | `( key1 key2`<br>`... )` | |
| (rmt:get-<br>key-val-<br>pairs<br>run-id) | | `#t=success/#f=fail` | `Works only if`<br>`the server is`<br>`still`<br>`reachable` |

# Chapter 20

# Test Plan

## 20.1  Tests

itemwait|33

rerun-downstream-item|20

rerunclean|20

fullrun|18

goodtests|18

kill-rerun|17

items-runconfigvars|16

ro_test|16

runconfig-tests|16

env-pollution|13

itemmap|11

testpatt_envvar|10

toprun|10

chained-waiton|8

skip-on-fileexists|8

killrun_preqfail|7

subrun|6

dependencies|5

itemwait-simple|4

rollup|4

end-of-run|3

killrun|3

listener|3

test2|3

testpatt|3

env-pollution-usecacheno|2

set-values|2 envvars|1 listruns-tests|1 subrun-usecases|1

# Chapter 21

# Megatest Internals