
Fuego Documentation

Release 1.5.8

Cogent Embedded, Tim Bird

Feb 23, 2023

CONTENTS:

1	Welcome to Fuego!	3
1.1	Introduction presentation	3
2	Getting Started	5
2.1	Where to download	5
3	Documentation	7
4	Resources	9
4.1	Mailing list	9
4.2	Presentations	9
5	Vision	11
6	Other Resources	13
6.1	Historical information	13
6.2	Related systems	13
6.3	Things to do	13
7	Index	15
7.1	About Fuego	15
7.2	Architecture	15
7.3	Install and First Test	25
7.4	Fuego Quickstart Guide	28
7.5	Installing Fuego	33
7.6	Adding a Board	40
7.7	Adding a toolchain	45
7.8	Adding test jobs to Jenkins	48
7.9	Adding views to Jenkins	50
7.10	Test variables	52
7.11	Jenkins User interface	56
7.12	Adding or Customizing a Distribution	67
7.13	Integration with ttc	70
7.14	Working with remote boards	72
7.15	Building Documentation	73
7.16	Fuego Developer Notes	74
7.17	License And Contribution Policy	84
7.18	Core interfaces	88
7.19	Adding a new test	95
7.20	Using Batch Tests	102
7.21	Parser module API	106

7.22	FAQ	110
7.23	Fuego naming rules	111
7.24	Artwork	115
7.25	Glossary	118
7.26	Raspberry Pi Fuego Setup	122
7.27	Using the qemuarm target	127
7.28	Variables	129
7.29	Dynamic Variables	136
7.30	Troubleshooting Guide	136
7.31	OSS Test Vision	139
7.32	Test Specs and Plans	145
7.33	parser.py	150
7.34	criteria.json	153
7.35	Coding Style	160
7.36	Testplan Reference	162
7.37	Indices and tables	165

Note: This documentation is being updated with material from the Fuego wiki (at fuegotest.org). Please be patient while this work is in progress.

WELCOME TO FUEGO!

Fuego is a test system specifically designed for embedded Linux testing. It supports automated testing of embedded targets from a host system, as its primary method of test execution.

Fuego consists of a host/target script engine, and over 100 pre-packages tests. These are installed in a docker container along with a Jenkins web interface and job control system, ready for out-of-the-box Continuous Integration testing of your embedded Linux project., and over 100 pre-packaged

The idea is that in the simplest case, you just add your board, select or install a toolchain, and go!

1.1 Introduction presentation

Tim Bird gave some talks introducing Fuego, at various conferences in 2016. The slides and a video are provided below, if you want to see an overview and introduction to Fuego.

The slides are here: [Introduction-to-Fuego-LCJ-2016.pdf](#), along with a [YouTube video](#). You can find more presentations about Fuego on our wiki at: <http://fuegotest.org/wiki/Presentations>.

GETTING STARTED

There are a few different ways to get started with Fuego:

1. Use the *Fuego Quickstart Guide* to get Fuego up and running quickly.
2. Or go through our *Install and First Test* tutorial to install Fuego and run a test on a single “fake” board. This will give you an idea of basic Fuego operations, without having to configure Fuego for your own board.
3. Work through the documentation for *Installation*

2.1 Where to download

Code for the test framework is available in 2 git repositories:

- <https://bitbucket.org/fuegotest/fuego/>
- <https://bitbucket.org/fuegotest/fuego-core/>

The fuego-core directory resides inside the fuego directory. But normally you do not clone that repository directly. It is cloned for you during the Fuego install process. See the *Fuego Quickstart Guide* or the *Installing Fuego* page for more information.

DOCUMENTATION

See the index below for links to the major sections of the documentation for Fuego. The major sections are:

- *Tutorials*
- *Installation and Administration*
- *User Guides*
- *Developer Resources*
- *API Reference*

RESOURCES

4.1 Mailing list

Fuego discussions are held on the fuego mailing list:

- <https://lists.linuxfoundation.org/mailman/listinfo/fuego>

Note that this is a new list (as of September 2016). Previously, discussions about Fuego (and its predecessor JTA) were held on the ltsi-dev mailing list:

- <https://lists.linuxfoundation.org/mailman/listinfo/ltsi-dev>

4.2 Presentations

A number of presentations have been given on the Fuego test framework, and related projects (such as its predecessor JTA, and a derivative project JTA-AGL).

See the [Presentations](#) page on the Fuego wiki for a list of presentations that you can read or view for more information about Fuego.

VISION

The purpose of Fuego is to bring the benefits of open source to the testing process.

It can be summed up like this:

Note: Do for testing what open source has done for coding

There are numerous aspects of testing that are still done in an ad-hoc and company-specific way. Although there are open source test frameworks (such as Jenkins or LAVA), and open source test programs (such as cyltest, LTP, linuxbench, etc.), there are lots of aspects of Linux testing that are not shared.

The purpose of Fuego is to provide a test framework for testing embedded Linux, that is distributed and allows individuals and organizations to easily run their own tests, and at the same time allows people to share their tests and test results with each other.

Historically, test frameworks for embedded Linux have been difficult to set up, and difficult to extend. Many Linux test systems are not easily applied in cross or embedded environments. Some very full frameworks are either not viewed as processor-neutral, and are difficult to set up, or are targeted at running tests on a dedicated group of boards or devices.

The vision of open source in general is one of sharing source code and capabilities, to expand the benefits to all participants in the ecosystem. The best way to achieve this is to have mechanisms to easily use the system, and easily share enhancements to the system, so that all participants can use and build on each others efforts.

The goal of Fuego is to provide a framework that any group can install and use themselves, while supporting important features like cross-compilation, host/target test execution, and easy test administration. Test administration consists of starting tests (both manually and automatically), viewing test results, and detecting regressions. Ease of use is critical, to allow testers to use tests that are otherwise difficult to individually set up, configure, and interpret the results from. It is also important to make it very easy to share tests (scripts, configuration, results parsing, and regression detection methods).

Some secondary goals of this project are the ability for 3rd parties to initiate or schedule tests on our hardware, and the ability to share our test results with others.

The use of Jenkins as the core of the test framework already supports many of the primary and secondary goals. The purpose of this project is to augment the Jenkins system to support embedded configurations of Linux, and to provide a place for centralized sharing of test configurations and collateral.

There is no such thing as a “Linux Test distribution”. Fuego aims to be this. It intends to provide test programs, a system to build, deploy and run them, and tools to analyze, track, and visualize test results.

For more details about a high-level vision of open source testing, please see [OSS Test Vision](#).

OTHER RESOURCES

6.1 Historical information

<http://elinux.org/Fuego> has some historical information about Fuego.

6.2 Related systems

See Other test systems for notes about other test frameworks and comparisons between Fuego and those other systems.

6.3 Things to do

Looking for something to do on Fuego? See the Fuego wiki for a list of projects, at: [Fuego To Do List](#)

7.1 About Fuego

Fuego is a test system specifically designed for embedded Linux testing. It supports automated testing of embedded targets from a host system, as it's primary method of test execution.

The quick introduction to Fuego is that it consists of a host/target script engine and over 100 pre-packaged tests. These are installed in a docker container along with a Jenkins web interface and job control system, ready for out-of-the-box Continuous Integration testing of your embedded Linux project.

The idea is that in the simplest case, you just add your board, a toolchain, and go!

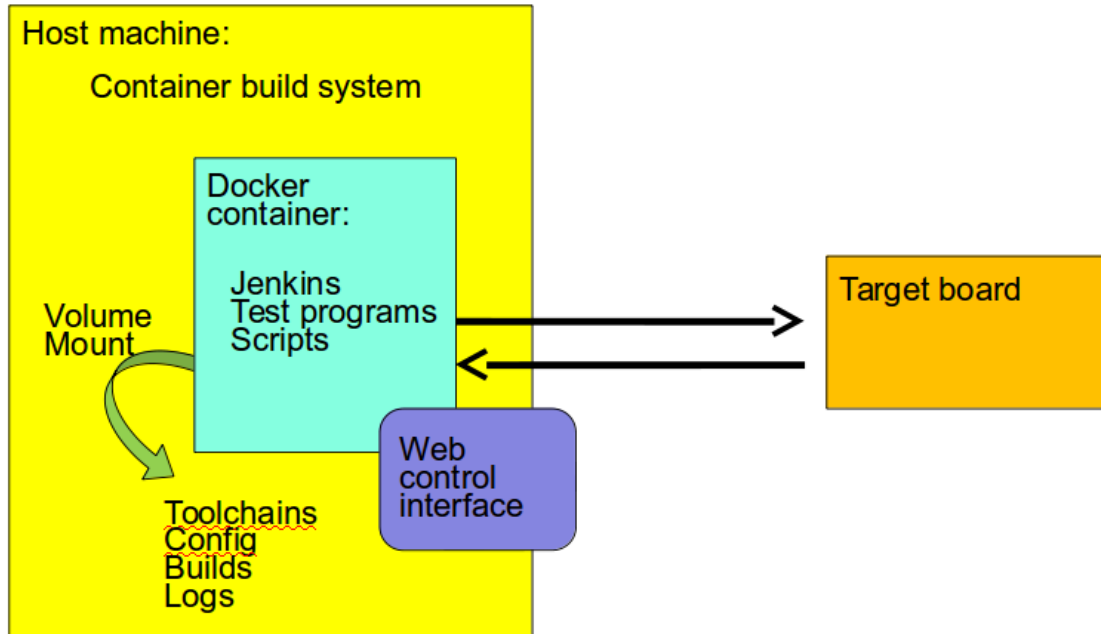
7.2 Architecture

Fuego consists of a continuous integration system, along with some pre-packaged test programs and a shell-based test harness, running in a Docker container.:

<pre>Fuego = (Jenkins + core scripts + pre-packaged tests) inside a container</pre>

Here's a diagram with an overview of Fuego elements:

Architecture Diagram



7.2.1 Major elements

The major elements in the Fuego architecture are:

- Host system
 - Container build system
 - Fuego container instance
 - * Jenkins continuous integration system
 - Web-based user interface (web server on port 8090)
 - Plugins
 - * Test programs
 - * Build environment (not shown in the diagram above)
 - * Fuego core system
- Target system
- Web client, for interaction with the system

7.2.2 Jenkins

The main interface for Fuego is provided by the Jenkins continuous integration system.

The basic function of Jenkins is to automatically launch test jobs, usually in response to changes in the software. However, it can launch test jobs based on a variety of triggers, including when a user manually schedules a test to run.

Jenkins is too big a system to describe in detail here, but it has many features and is very popular. It has an ecosystem of plugins for all kinds of extended functionality, such as integration with different source code management systems, results plotting, e-mail notifications of regressions, and more.

Fuego installs several plugins that are used by various aspects of the system.

Jenkins is used to:

- Start tests
- Schedule tests for automatic execution
- Shows test results (particularly over time)
- Flag regressions in test results

Note that the interface between Jenkins and the test programs is provided by a set of scripts (one per test, along with a set of scripts that comprise the core of the system) written in shell script language.

The interface between Jenkins and these core scripts is documented at Core interfaces.

This overall architecture means that when items are added into the system (for example boards, toolchains, or tests), information has to be supplied to both systems (the Jenkins system and the core script system).

7.2.3 Pre-packaged tests

Fuego contains over 100 pre-packaged tests, ready for you to start testing “out-of-the-box”. There are tests of individual programs or features, such as ‘iputils’ or ‘pmqtest’, as well as several benchmarks in the areas of CPU performance, networking, graphics, and realtime. Fuego also includes some full test suites, like LTP (Linux Test Project). Finally, Fuego includes a set of selftests, to validate board operation or core Fuego functionality.

7.2.4 Abstraction scripts

Fuego uses a set of shell script fragments to support abstractions for

- Building test programs from source,
- Deploying them to target (installing them)
- Executing the tests
- Copying files to and from the target
- Reading the test log
- Parsing the log to determine pass or fail conditions for tests
- Parsing the log for results to display in charts

7.2.5 Container

By default, Fuego runs inside a Docker container. This provides two benefits:

- It makes it easy to run the system on a variety of different Linux distributions
- It makes the build environment for the test programs consistent

7.2.6 Hardware configuration

Fuego supports testing of embedded Linux by fully supporting a host/target configuration for building, deploying and executing tests.

Many Linux test systems assume that the system-under-test is a full desktop or server system, with sufficient horsepower to build tests and run them locally. Fuego assumes the opposite - that embedded targets will be underpowered and may not have the normal complement of utilities and tools available for performing tests

7.2.7 Different objects in Fuego

It is useful to give an overview of the major objects used in Fuego, as they will be referenced many times:

Fuego core objects:

- board - a description of the device under test
- test - materials for conducting a test
- spec - one or more sets of variables for describing a test variant
- plan - a collection of tests, with additional test settings for their execution
- run - the results from a individual execution of a test on a board

Jenkins objects:

- node - the Jenkins object corresponding to a Fuego board
- job - a Jenkins object corresponding to a combination of board, spec, and test
- build - the test results, from Jenkins perspective - corresponding to a Fuego 'run'

There are both a front-end and a back-end to the system, and different names are used to describe the front-end and back-end objects used by the system, to avoid confusion. In general, Jenkins objects have rough counterparts in the Fuego system:

Jenkins object	corresponds to fuego object
node	board
job	test
build	run

7.2.8 Jenkins operations

This section explains how Jenkins works as part of Fuego.

- When the a job is initiated, Jenkins starts a slave process to run the test that corresponds to that job
- Jenkins records stdout from slave process
- The slave (slave.jar) runs a script specified in the config.xml for the job
 - This script sources functions from the scripts and overlays directory of Fuego, and does the actual building, deploying and test executing
 - Also, the script does results analysis on the test logs, and calls the post_test operation to collect additional information and clean up after the test
- While a test is running, Jenkins accumulates the log output from the generated test script and displays it to the user (if they are watching the console log)
- Jenkins provides a web UI for browsing the nodes, jobs, and test results (builds), and displaying graphs for benchmark data

7.2.9 Fuego operations

This section explains how the Fuego core system works to execute tests and analyze results.

7.2.10 Test execution

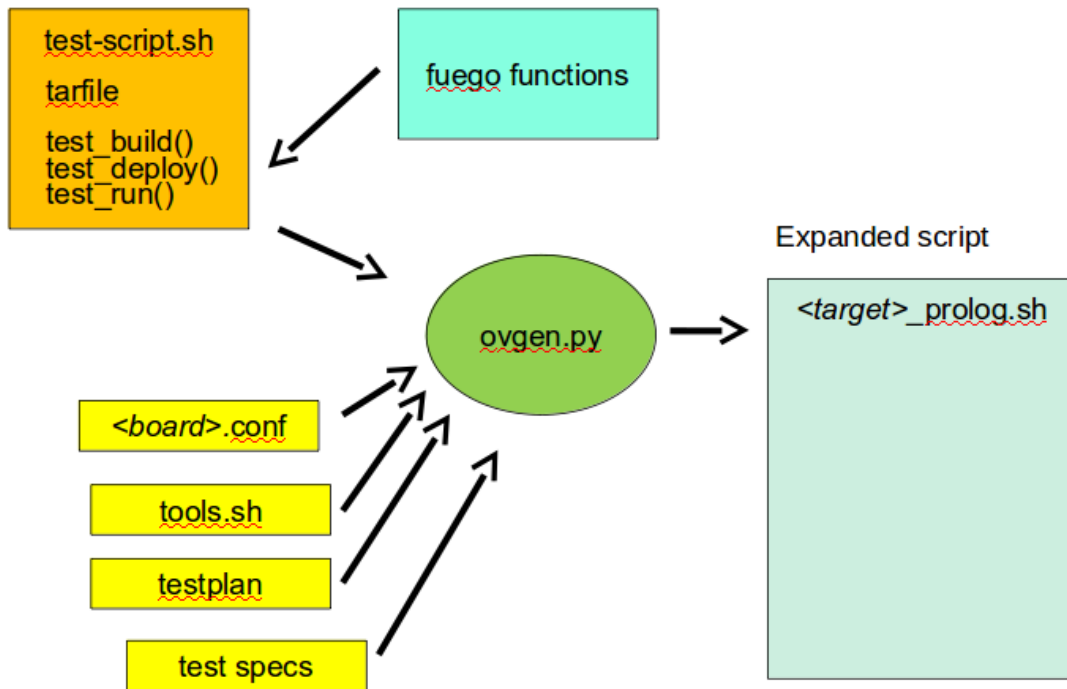
- Each test has a base script, that defines a few functions specific to that test (see below)
- Upon execution, this base script loads additional test variables and function definitions from other files using something called the overlay generator
- The overlay generator creates a script containing test variables for this test run
 - The script is created in the log directory for the test
 - The script is called prolog.sh
 - The overlay generator is called ovgen.py
- The base script (with the test variable script sourced into it) runs on the host, and uses Fuego functions to perform different phases of the test
- For a flow graph of normal test execution see: *[test execution flow outline](#)*

7.2.11 Test variable file generation

- The generator takes the following as input:
 - environment variables passed by Jenkins
 - board file for the target (specified with NODE_NAME)
 - tools.sh (vars from tools.sh are selected with TOOLCHAIN, from the board file)
 - the distribution file, and (selected with DISTRIB)
 - test specs for the test

The generator creates a file containing test variables, called `prolog.sh` and it is placed in the the log directory for the test (also referred to as the “run” directory). This generation happens on the host, inside the docker container. Besides variable definitions, the test variable file has functions which are called by the test during test execution.

Base script



Input

- input descriptions:
 - the board file has variables defining attributes of the board, like the toolchain, network address, method of accessing the board, etc.
 - `tools.sh` has variables which are used for identifying the toolchain used to build binary test programs
 - * it uses the `TOOLCHAIN` variable to determine the set of variables to define
 - a `testplan` lists multiple tests to run
 - * it specifies a test name and spec for each one
 - * a spec files hold the a set of variable declarations which are used by the tests themselves. These are put into environment variables on the target.
- `ovgen.py` reads the plans, board files, distrib files and specs, and produces a single `prolog.sh` file that has all the information for the test
- Each test in the system has a Fuego shell script, called `fuego_test.sh`
- Most (but not all) tests have an additional test program
 - this program is executed on the board (the device under test)
 - it is often a compiled program, or set of programs

- it can be a simple shell script
 - it is optional - sometime the base script can execute the needed commands for a test without an additional program placed on the board
- the base script declares the tarfile for the test, and has functions for: `test_build()`, `test_deploy()` and `test_run()`
 - the test script is run on host (in the container)
 - * but it can include commands that will run on the board
 - tarball has the tarfile
 - `test_build()` has commands (which run in the container) to compile the test program
 - `test_deploy()` has commands to put the test programs on the target
 - `test_run()` has commands to define variables, execute the actual test, and log the results.
- the test program is run on the target
 - this is the actual test program that runs and produces a result

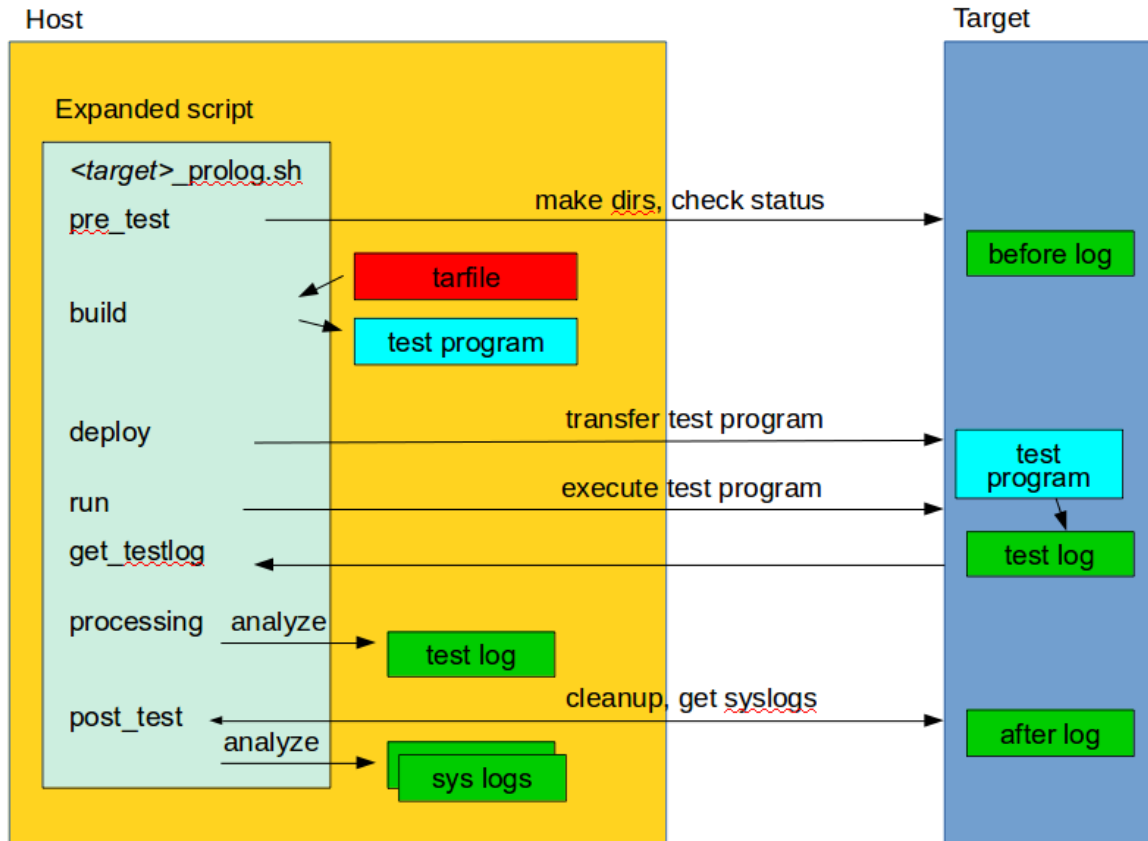
7.2.12 Fuego test phases

A test execution in fuego runs through several phases, some of which are optional, depending on the test.

The test phases are:

- `pre_test`
- `build`
- `deploy`
- `run`
- `fetch`
- `processing`
- `post_test`

Each of these are described below the diagram.



pre_test

The `pre_test` phase consists of making sure the target is alive, and preparing the workspace for the test. In this phase test directories are created, and the firmware (a string describing the software on the target) are collected.

The ‘before’ syslog is created, and filesystems are synced and buffer caches dropped, in preparation for any filesystem tests.

If the function “`test_pre_check`” is defined in the base script, it is called to do any `pre_test` operations, including checking to see if required variables are set, programs or features are available on the host or target board, or whether any other test dependencies are met.

build

During this phase, the test program source is installed on the host (inside the container), and the software for the test is actually built. The toolchain specified by `PLATFORM` is used to build the software.

This phase is split into multiple parts:

- `pre_build` - build workspace is created, a build lock is acquired and the tarball is unpacked
 - `unpack` is called during `pre_build`
- `test_build` - the `test_build` function from `fuego_test.sh` is called
 - usually this consists of ‘`make`’, or ‘`configure ; make`’
- `post_build` - (empty for now)

deploy

The purpose of this phase is to copy the test programs, and any required supporting files, to the target.

This consists of 3 sub-phases:

- `pre_deploy` - `cd`'s to the build directory
- `test_deploy` - the base script's `'test_deploy'` function is called.
 - Usually this consists of tarring up needed files, copying them to the target with `'put'`, and then extracting them there
 - Items should be placed in the directory `$BOARD_TESTDIR/fuego.$TESTDIR/` directory on the target
- `post_deploy` - removes the build lock

run

In this phase the test program on the target is actually executed.

This executes the `'test_run'` function defined in the base script for the test, which can consist of anything. Usually, however, it runs the test program with any needed parameters (as specified by the test specs and test plans).

The test execution is usually performed by calling the `'report'` function, which collects the standard out from the command execution on the target, and saves that as the testlog for the test. Note that the testlog is saved on the target, but not yet transferred to the host, yet.

post_test

In this phase, the test log is retrieved (fetched) from the target and stored on the host. Also in this phase, the board is “cleaned up”, which means that test directories and logs are removed on the target board, and any leftover processes related to this test that are running on the board are stopped.

processing

In the processing phase of the test, the results from the test log are evaluated. The `test_processing` function of the base test script is called.

For functional tests:

Usually, this phase consists of one or more calls to `'log_compare'`, to determine if a particular string occurs in the testlog. This phase determines whether the test passed or failed, and the base test script indicates this (via its exit code) to the Jenkins interface.

For benchmarking tests:

This phase consists of parsing the testlog, using `parser.py`, and also running `dataload.py` to save data for plot generation.

Also, a final analysis is done on the system logs is done in this step (to detect things like Kernel Ooopses that occurred during the test).

phase relation to base script functions

Some of the phases are automatically performed by Fuego, and some end up calling a routine in the base script (or use data from the base script) to perform their actions. This table shows the relation between the phases and the data and routines that should be defined in the base script.

It also shows the most common commands utilized by base script functions for this phase.

phase	relationship to base script	common operations
pre_test	calls 'test_pre_check'	assert_define, is_on_target, check_process_is_running
build	uses the 'tarfile' definition, calls 'test_build'	patch, configure, make
deploy	Calls 'test_deploy'	put
run	calls 'test_run'	cmd, report, report_append
get_testlog	(none)	
process-ing	calls 'test_processing'	log_compare
post_test	calls 'test_cleanup'	kill_procs

Other scripts and programs

- `parser.py` - for parsing test results
- `criteria.json` - for analyzing test results

A test might also include a file called `parser.py`. In fact, every benchmark test should have one. This file is a python module which is run to extract results and data values from the log.

This script is run inside the docker container, after a test is finished. The Fuego log parsing system (in python) loads this module as part of test processing.

A benchmark program measures some attribute of the system during a test, and produces one or more values called 'metrics'. These values are emitted by the benchmark test into the test log, and the Fuego parser retrieves these values and uses them to evaluate the pass/fail status of the benchmark. These values are saved as part of the test results, and are used by plotting software to show charts of test results in the Jenkins interface.

Tests may also include a file called `criteria.json` which is used to determine whether test results constitute a pass or fail result. For example, for benchmark tests, the system can collect a number from the test program, but it is up to the system to determine whether that number represents an acceptable value (pass), or a failure or regression (fail). The `criteria.json` file has data about metric thresholds, for benchmark tests, and about test results that can be ignored, for functional tests, to allow for automating this results processing.

7.2.13 Data Files

There are data files with definitions for several things in the system.

The Jenkins interface needs to know about boards, running test processes (slaves), test definitions, and test results.

The fuego core needs to know about test definitions, boards, platforms (SDKS), test plans, and test specs.

The core executes the test script for a test, executing the test phases in sequence: build the test program, bundle the test programs for the target, deploy them, execute the test, then post-process the the test.

The base shell script should:

- build the test program

- deploy the test bundle to the target
- execute the tests
- read the log data from the test

The base shell script can handle host/target tests (because it runs on the host). That is, tests that involve actions on both the host and target.

To add a new test, the user defines several files and puts them into `/fuego-core/engine/tests`

The `ftc` command uses this directory and the directory `/fuego-ro/boards` to show available boards and tests on the command line. Then a user can populate the nodes and jobs in Jenkins, using `ftc` commands. Each test has to have a front-end entry (a Jenkins job definition) to allow Jenkins to execute it. This front-end entry specifies the board, spec and base script for the test.

7.2.14 Roles

Human roles:

- test program author - person who creates a new standalone test program
- test integrator - person who integrates a standalone test into fuego
- fuego developer - person who modifies Fuego (including the Fuego system scripts or Jenkins) to support more test scenarios or additional features
- tester - person who executes tests and evaluates results

7.3 Install and First Test

This tutorial has some short setup instructions if you just want to get a taste of what Fuego is like. This allows you to experiment with Fuego and try out some tests to see what it looks like and how it works, without investing a lot of time (well, except for the first container build).

In this configuration, we will show you how to install Fuego and run a test on a ‘docker’ board, which is the docker container where Fuego itself is running, on your host machine.

Obviously, this is not useful for testing any real hardware. It is intended only as a demonstration of Fuego functionality.

For instructions to set up a real board, try the Fuego Quickstart Guide or the [Installing Fuego](#) page.

7.3.1 Overview

An overview of the steps is:

1. Install pre-requisite software
2. Download the Fuego repository
3. Build your Fuego container
4. Start the container
5. Add the ‘docker’ board to Jenkins
6. Add some sample tests
7. Access the Jenkins interface
8. Run a test

These steps are described below.

7.3.2 Step details

To install and run Fuego, you need to have git and docker installed on your system.

On Ubuntu, try the following commands:

```
$ sudo apt install git docker.io
```

To download Fuego, and build and start the container, type the following commands at a Linux shell prompt:

```
$ git clone https://bitbucket.org/fuegotest/fuego.git
$ cd fuego
$ ./install.sh
$ ./start.sh
```

The third step (with `./install.sh`) will take some time - about 45 minutes on an average Linux machine. You might want to go make yourself a sandwich, (or go watch the [Fuego introduction video](#)). This step is building the “Fuego” distribution of Linux (based on Debian) and putting it into the Fuego docker container. You will also need a connection to the Internet with fairly decent bandwidth.

When you run the ‘start.sh’ script, the terminal will be placed at a shell prompt, as the root user inside the docker container. The container will run until you exit this shell. You should leave it running for the duration of your testing.

The next steps populate the Jenkins system objects used for testing:

At the shell prompt inside the container type the following:

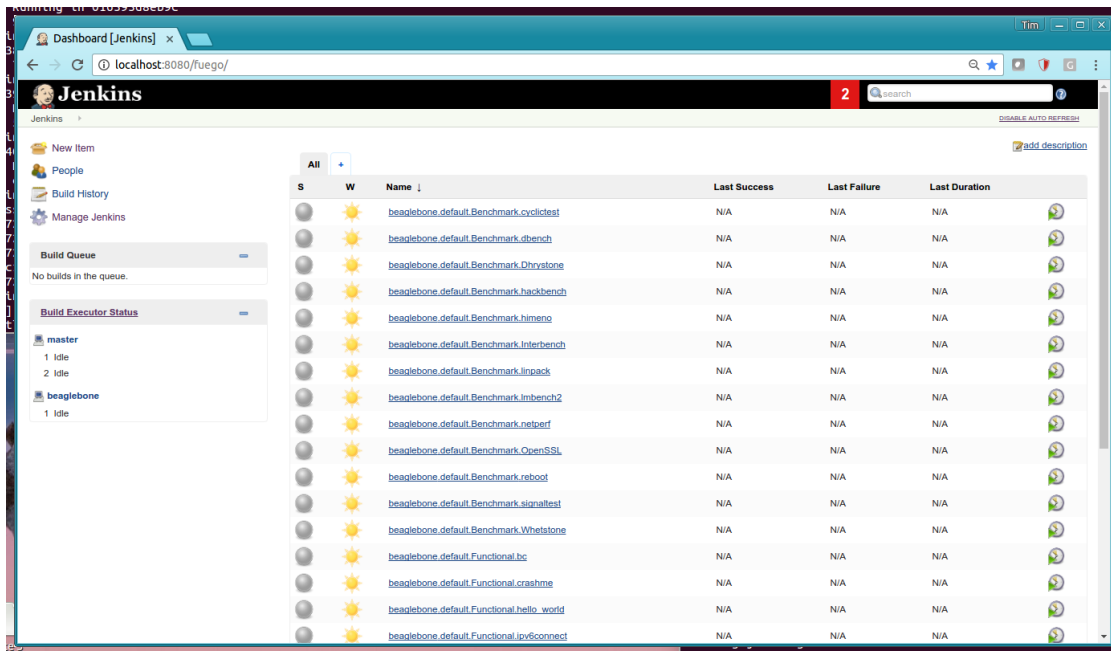
```
# ftc add-node -b docker
# ftc add-jobs -b docker -t Functional.batch_smoketest
```

This will add the ‘docker’ node in the Jenkins interface, and a small set of tests. The ‘docker’ node simulates an embedded “board” using the Fuego docker container itself. The “smoketest” batch test has about 20 tests that exercise a variety of features in a Linux system. After running these commands, a set of jobs will appear in the Jenkins interface.

```
$ firefox http://localhost:8090/fuego
```

To access the Fuego interface (Jenkins) you can use any browser - not just Firefox. By default the Fuego interface runs on your host machine, on port 8090, with URL path “/fuego”.

In your browser, you should see a screen similar to the following:



7.3.3 Run a test

Manually start a test

To run a job manually, do the following:

- Go to the Jenkins dashboard (in the main Jenkins web page),
- Select the job (which includes the board name and the test name)
- Click “Build job” (Jenkins refers to running a test as “building” it.)

A few very simple jobs you might start with are:

- Functional.hello_world
- Benchmark.Drhystone

You can also start a test manually by clicking on the circle with a green triangle, on the far right of the line with the job name, in the Jenkins dashboard.

When you run a test, the test software is built from source, sent to the machine (in this case the Fuego docker container), and executed. Then the results are collected, analyzed, and displayed in the Jenkins interface.

Look at the results

When the test has completed, the status will be shown by a colored ball by the side of the test in the dashboard. Green means success, red means failure, and grey means the test did not complete (it was not run or it was aborted).

You can get details about the test by clicking on the links in the history list. You can see the test log (the output from the test program), by clicking on the “testlog” link. You can see the steps Fuego took to execute the test by clicking on the “console log” link on the job page. And you can see the formatted results for a job, and job details (like start time, test information, board information, and results) in the test’s ‘run.json’ file.

7.3.4 Conclusions

Hooray! You have completed your first Fuego test. Although this was not a test on real hardware, you (and Fuego) have completed a lot of stuff behind the scenes. You have:

- Downloaded your own distribution of Fuego and installed it in a docker container
- Added a fake “board” to Jenkins
- Added tests to run on the board
- Executed a test

During this last step, Fuego did the following:

- Built the test program from source code
- Downloaded the test to the “board”
- Executed the test
- Retrieved the test log from the board
- Analyzed the log, and formatted results for display in Jenkins

Whew! That’s a lot of work. And all you had to do (after initial installation) was click a button.

7.3.5 What do do next?

In order to use Fuego in a real Continuous Integration loop, you need to do a few things:

- Configure Fuego to work with your own board or product
- Select the set of tests you would like to run on your board
- Customize benchmark thresholds and functional baselines for those tests, for your board
- Configure Fuego jobs to be triggered after new software is installed on the board

Fuego does not currently have support for installing new system software (the kernel and root filesystem) on boards itself. This is something you need to automate outside of Fuego, if you plan to use Fuego in your CI loop for system software.

Usually, Fuego users create their own Jenkins job which provisions the board (installs the kernel and/or root filesystem for their chosen Linux distribution), and then triggers Fuego jobs, after the new software is installed on the board.

See further instructions see the Fuego [Quickstart Guide](#), [Adding a Board](#), [Adding a toolchain](#) or the [Installing Fuego](#) page.

7.4 Fuego Quickstart Guide

Running tests from Fuego on your hardware can be accomplished in a few simple steps.

Note: This is the Quickstart Guide. More detailed instructions can be found at [Installing Fuego](#).

7.4.1 Overview

The overview of the steps is:

1. install pre-requisite software
2. download the fuego repository
3. build your fuego container
4. start the container
5. access the interface
6. add your board to fuego
7. run a test

These steps are described below.

7.4.2 Install pre-requisite software

To retrieve the fuego software and create the docker image for it, you need to have git and docker installed on your system.

On Ubuntu, try the following commands::

```
$ sudo apt install git docker.io
```

7.4.3 Download, build, start and access

To accomplish the last 6 steps, do the following from a Linux command prompt::

```
$ git clone https://bitbucket.org/fuegotest/fuego.git
$ cd fuego
$ ./install.sh
$ ./start.sh
$ firefox http://localhost:8090/fuego
```

The fourth step (with `./install.sh`) will take some time - about 45 minutes on my machine. This is the main step that builds the Fuego docker container.

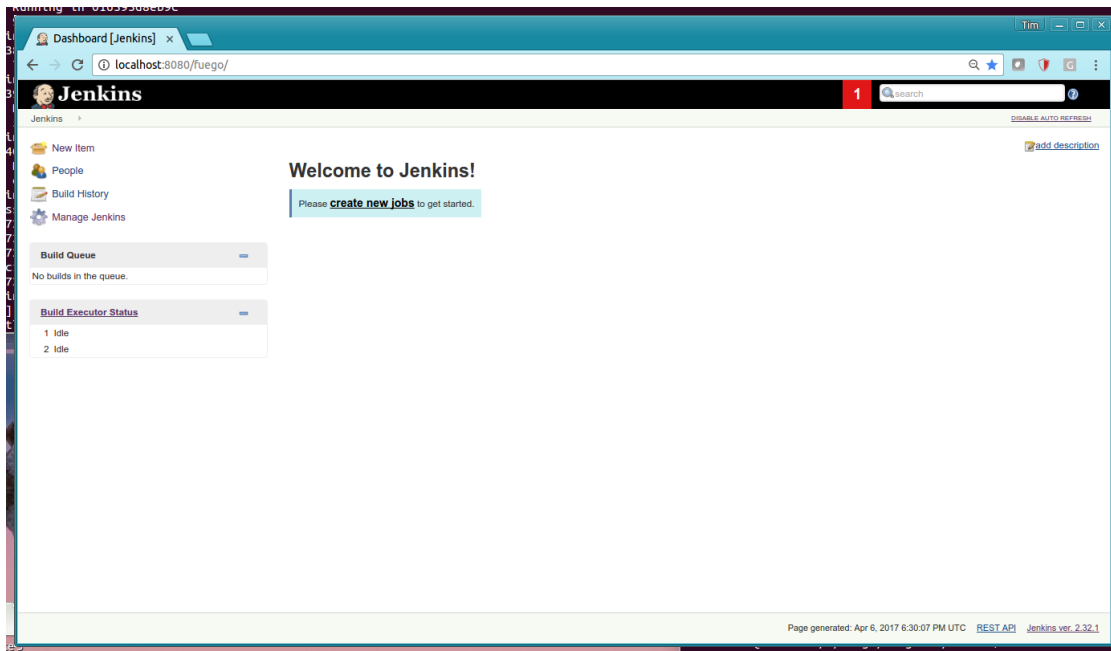
When you run the `'start.sh'` script, the terminal where this is run will be placed at a shell prompt, as the root user, inside the docker container. The container will run until you exit this shell. You should leave it running for the duration of your testing.

Note: If you are experimenting with the unreleased version of Fuego in the `'next'` branch, then please replace the `'git clone'` command in the instructions above with this (i.e. add `'-b next'`):

```
$ git clone -b next https://bitbucket.org/fuegotest/fuego.git
```

On the last step, to access the Fuego interface you can use any browser - not just Firefox. By default the Fuego interface runs on your host machine, on port 8090, with URL path `"/fuego"`.

In your browser, you should see a screen similar to the following:



We will now add items to Fuego (and this screen) so you can begin testing.

7.4.4 Add your board to fuego

To add your own board to Fuego, there are three main steps:

- 1. create a test directory on the target
- 2. create a board file (on the host)
- 3. add your board to the Jenkins interface

You can find detailed instructions for adding a board at: [Adding a Board](#)

However, here is a quick list of steps you can do to add a your own board, and a sample ‘docker’ board to Fuego:

Create a test directory on your board

Login to your board, and create a directory to use for testing::

```
$ ssh root@your_board
<board>$ mkdir /home/a
<board>$ exit
```

If not using ssh, use whatever method you normally use to access the board.

Create board file

Now, create your board file. The board file resides in <fuego-dir>/fuego-ro/boards, and has a filename with the name of the board, with the extension “.board”.

Do the following:

```
$ cd fuego-ro/boards
$ cp template-dev.board myboard.board
$ vi myboard.board
```

Edit the variables in the board file to match your board. Most variables can be left alone, but you will need to change the IPADDR, TOOLCHAIN and ARCHITECTURE variables, and set the BOARD_TESTDIR to the directory you just created above.

For other variables in the board file, or specifically to use a different transport than SSH, see more complete instructions at: [Adding a Board](#)

Add boards to the Jenkins interface

Finally, add the board in the Jenkins interface.

In the Jenkins interface, boards are referred to as “Nodes”.

At the container shell prompt, run the following command:

- (container prompt)\$ ftc add-nodes -b myboard docker

This will add your board as a node, as well as a ‘docker’ node in the Jenkins interface.

7.4.5 Install a toolchain

If you just wish to run experiment with Fuego, without installing your own board, you can use the existing ‘docker’ board. This will run the tests inside the docker container on your host machine. This requires little setup, and is intended to let people try Fuego to see how the interface and tests work, without having to set up their own board.

If you are running an ARM board with a Debian-based distribution on it, you can install the Debian ARM cross-compilers into the docker container with the following command (inside the container):

- (container prompt)\$ /fuego-ro/toolchains/install_armhf_toolchain.sh

If you are installing a some other kind of board (different architecture, different root filesystem layout, or different shared library set), you will need to install a toolchain for your board inside the docker container.

Please follow the instructions at: [Adding a toolchain](#) to do this.

7.4.6 Now select some tests

In order to execute tests using the Jenkins interface, you need to create Jenkins “jobs” for them. You can do this using the ‘ftc add-jobs’ command.

These commands are also executed at the shell prompt in the docker container.

You can add jobs individually, or you can add a set of jobs all at once based on something called a ‘testplan’. A testplan is a list of Fuego tests with some options for each one. You can see the list of testplans in your system with the following command:

- (container prompt)\$ ftc list-plans

To create a set of jobs for the ‘docker’ board on the system, do the following:

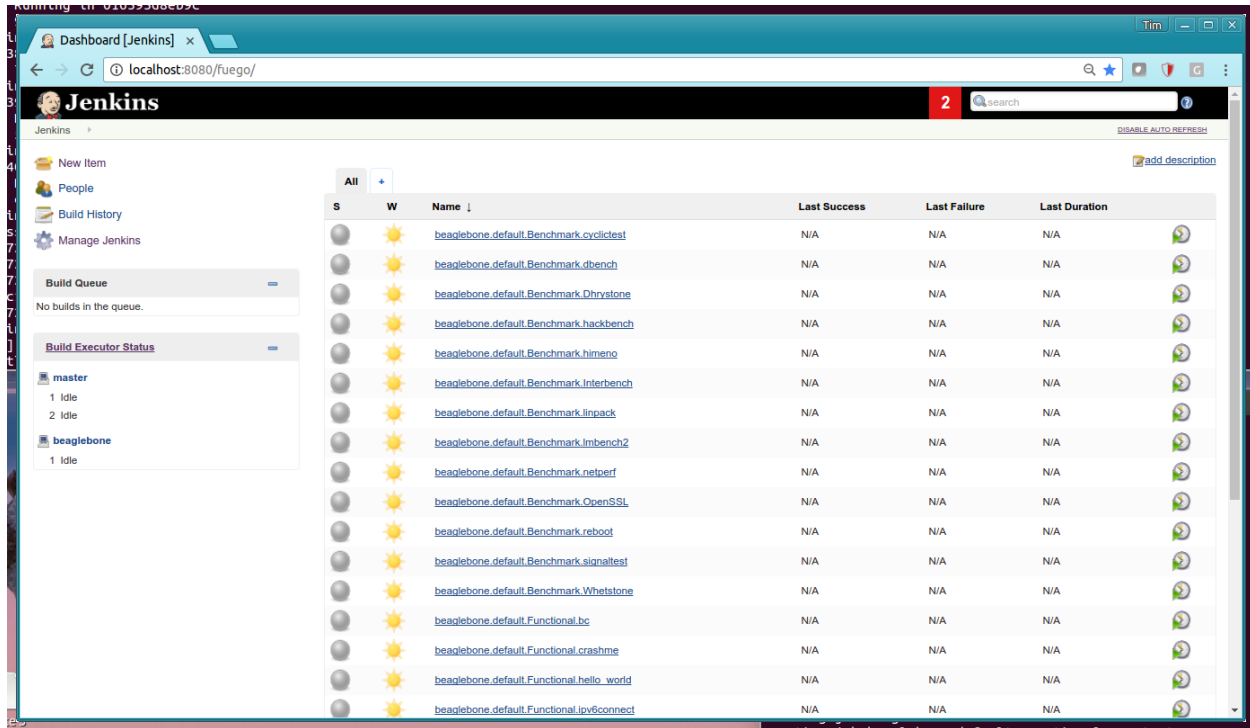
- (container prompt)\$ `ftc add-jobs -b docker -p testplan_docker`

To create a set of jobs for your own board (assuming you called it ‘myboard’), do the following:

- (container prompt)\$ `ftc add-jobs -b myboard -p testplan_smoketest`

The “smoketest” testplan has about 20 tests that exercise a variety of features in a Linux system. After running these commands, a set of jobs will appear in the Jenkins interface.

Once this is done, your Jenkins interface should look something like this:



7.4.7 Run a test

To run a job manually, you can do the following:

- Go to the Jenkins dashboard (in the main Jenkins web page),
- Select the job (which includes the board name and the test name)
- Click “Build job” (Jenkins refers to running a test as “building” it.)

You can also click on the circle with a green triangle, on the far right of the line with the job name, in the Jenkins dashboard.

When the test has completed, the status will be shown by a colored ball by the side of the test in the dashboard. Blue means success, red means failure, and grey means the test did not complete (was not run or was aborted). You can get details about the test run by clicking on the link in the history list.

7.4.8 Additional Notes

Other variables in the board file

Depending on the test you want to run, you may need to define some other variables that are specific to your board or the configuration of the filesystem on it. Please see [Adding a Board](#) for detailed instructions and a full list of variables that may be used on the target.

the Jenkins interface

See Jenkins User Interface for more screenshots of the Jenkins web interface. This will help familiarize you with some of the features of Jenkins, if you are new to using this tool.

7.4.9 Troubleshooting

If you have problems installing or using Fuego, please see our Troubleshooting Guide

7.5 Installing Fuego

This page describes the steps to install Fuego on your Linux machine. It includes detailed descriptions of the operations, for both users and developers.

Tip: If you are interested in a quick outline of steps, please see the [Fuego Quickstart Guide](#) instead.

7.5.1 Overview

The overview of the steps is:

1. Install pre-requisite software
2. Download the Fuego repository
3. Build your Fuego container
4. Start the container
5. Access the Jenkins interface

7.5.2 Install pre-requisite software

To retrieve the Fuego software and create the Docker image for it, you need to have **git** and **Docker** installed on your system.

On Ubuntu, try the following commands:

```
$ sudo apt-get install git
$ sudo apt-get install docker.io
```

These commands may be different for other distributions of Linux (such as Fedora, RedHat, CentOS, Mint, etc.)

7.5.3 Overview of remaining steps

Steps 2 through 5 of the installation can be performed with the following Linux commands:

```
$ git clone https://bitbucket.org/fuegotest/fuego.git
$ cd fuego
$ ./install.sh
$ ./start.sh
$ firefox http://localhost:8090/fuego
```

These steps and commands will be described in the sections that follow.

7.5.4 Install the Fuego repositories

The Fuego system is contained in 2 git repositories. One repository is called `fuego` and the other is called `fuego-core`. The `fuego-core` repository is installed inside the `fuego` directory, at the top level of that repository's directory structure. This is done automatically during the install of Fuego. You do not need to clone the `fuego-core` repository manually yourself.

The reason to have two repositories is that they hold different pieces of the Fuego system, and this allows for them to be upgraded independently of each other.

The repositories are hosted on `bitbucket.org`, under the the `fuegotest` account.

Fuego repository

The `fuego` repository has the code and files used to build the Fuego docker container. It also has the `fuego-ro` directory, which has board definition files, various configuration files, miscellaneous scripts, and other items which are used by Fuego for container management or other purposes.

Fuego-core repository

The `fuego-core` repository has the code which implements the core of the Fuego test execution engine, as well as the pre-packaged tests included with the system. This includes the overlay generator, the results parser, the Fuego shell function library, the directory of tests, and the main Fuego command line tool `ftc`.

Downloading the repository

You can use `git clone` to download the main Fuego repository, like so:

```
$ git clone https://bitbucket.org/fuegotest/fuego.git
$ cd fuego
```

After downloading the repositories, switch to the `fuego` directory, as shown in the example.

Note that these `git` commands will download the 'master' branch of the repository, which is the current main released version of Fuego.

Downloading a different branch

If you are experimenting with an unreleased version of Fuego in the ‘next’ branch, then please replace the ‘git clone’ command in the instructions above with these:

```
$ git clone -b next https://bitbucket.org/fuegotest/fuego.git
$ cd fuego
```

This uses `-b next` to indicate a different branch to check out during the clone operation.

7.5.5 Create the Fuego container

The third step of the installation is to run `install.sh` to create the Fuego docker container. While in the `fuego` directory, run the script from the current directory, like so:

```
$ ./install.sh
```

`install.sh` uses Docker and the Dockerfile in the `fuego` directory to create a Docker container with the Fuego Linux distribution.

This operation may take a long time. It takes about 45 minutes on my machine. This step assembles a nearly complete distribution of Linux, from binary packages obtained from the Internet.

This step requires Internet access. You need to make sure that you have proxy access to the Internet if you are behind a corporate firewall.

Please see the section “Alternative Installation Configuratons” below for other arguments to `install.sh`, or for alternative installation scripts.

Fuego Linux distribution

The Fuego Linux distribution is a distribution of Linux based on Debian Linux, with many additional packages and tools installed. These additional packages and tools are required for aspects of Fuego operation, and to support host-side processes and services needed by the tests included with Fuego.

For example, the Fuego distribution includes:

- the **Jenkins** continuous integration server
- the `netperf` server, for testing network performance.
- the `ttc` command, which is a tool for board farm management
- the python `jenkins` module, for interacting with Fuego’s Jenkins instance
- and many other tools, programs and modules used by Fuego and its tests

Fuego commands execute inside the Fuego docker container, and Fuego operations initiate in the container, but may access hardware (such as USB ports, networking, or serial ports) that are outside the container.

Configuring for ‘privileged’ hardware access

In many configurations, Fuego can perform its operations using only network operations. However, depending on the configuration of your boards, or your lab hardware, and the relationship between your host and target computers used for testing, you may need to access other hardware on your host machine.

To do that, you can create a ‘privileged’ Fuego container, using the `--priv` options with `install.sh`:

```
$ ./install.sh --priv
```

Customizing the privileged container

Note that using `--priv` causes `install.sh` to use a different container creation script. Normally (in the non `--priv` case), `install.sh` uses `fuego-host-scripts/docker-create-container.sh`. When `--priv` is used, Fuego uses `fuego-host-scripts/docker-create-usb-privileged-container.sh`.

This latter script (`docker-create-usb-privileged-container.sh`) can be edited, before running `install.sh`, to change the set of hardware devices that the Docker container will have privileged access to.

This is done by adding more bind mount options to the `docker create` command inside this script. Explaining exactly how to do this is outside the scope of this documentation. Please see documentation and online resources for the Docker system for information about this.

The script currently creates bind mounts for:

- `/dev/bus/usb` - USB ports, and newly created ports
- `/dev/ttyACM0` - serial port 0
- `/dev/ttyACM1` - serial port 1
- `/dev/serial` - general serial ports, and newly created ports

If you experience problems with Fuego accessing hardware on your host system, you may need to build the Fuego docker container using additional bind mounts that are specific to your configuration. Do so by editing `docker-create-used-privileged-container.sh`, removing the old container, and re-running `./install.sh --priv` to build a new container with the desired privileges.

Using an different container name

By default, `install.sh` creates a Docker image called `fuego` and a Docker container called `fuego-container`. There are some situations where it is desirable to use different names. For example, having different container names is useful for Fuego self-testing. It can also be used to do A/B testing when migrating from one release of Fuego to the next.

You can provide a different name for the Fuego image and container, by supplying one on the command line for `install.sh`, like so:

```
$ ./install.sh my-fuego
```

This would create a Docker image named `my-fuego` and a Docker container named `my-fuego-container`

7.5.6 Start the Fuego container

To start the Fuego docker container, use the `start.sh` script.

```
$ ./start.sh
```

Using a different container name

By default, `start.sh` will start the container named `fuego-container` (which is the default Fuego docker container name). However, if you created a different container name, you can specify the name on the command line, like so:

```
$ ./start.sh my-fuego-container
```

When you run the `start.sh`, the terminal where the script is run will be placed at a shell prompt inside the Docker container. The session will be logged in as the root user inside the container. The container will run until you exit this top-level shell. Therefore, you should leave it (the shell and the terminal that your ran `start.sh` from) running for the duration of your testing.

7.5.7 Access the Fuego Jenkins web interface

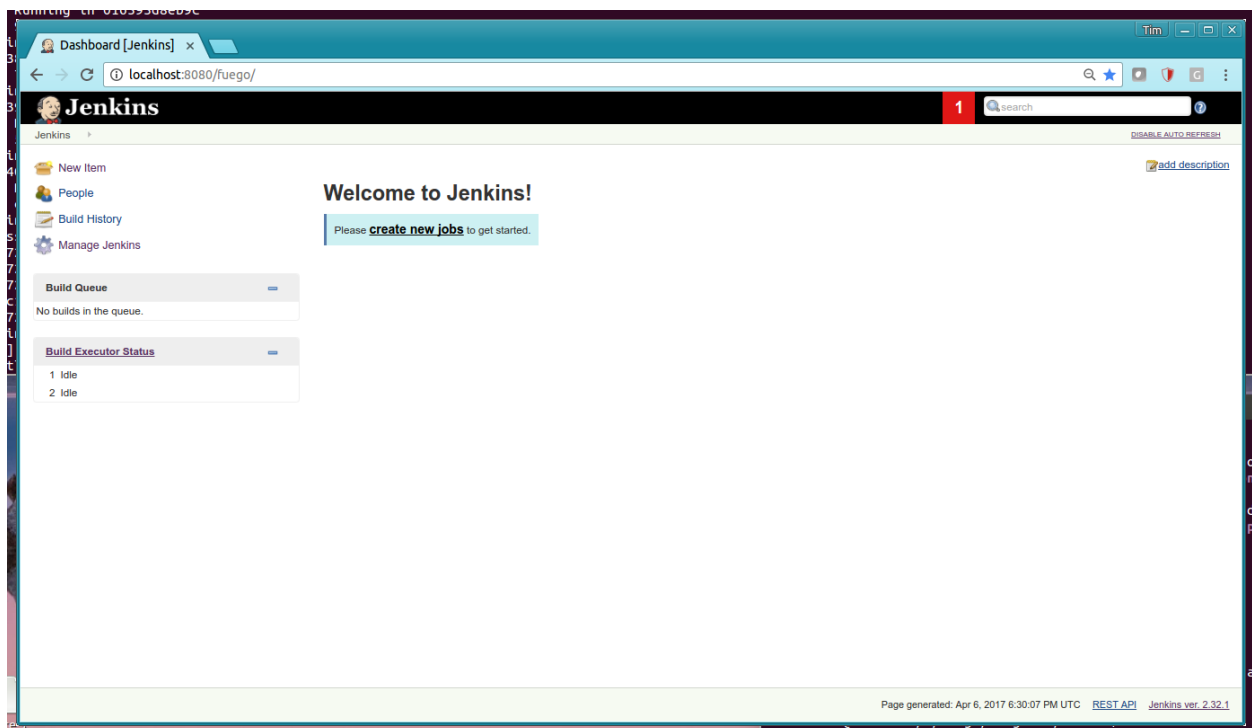
Fuego includes a version of Jenkins and a set of plugins as part of its system. Jenkins is running inside the Fuego docker container. By default the Fuego Jenkins interface runs on port `8090`, with an URL path `/fuego`.

Here is an example showing use of firefox to access the Jenkins interface with Fuego

```
$ firefox http://localhost:8090/fuego
```

To access the Fuego interface you can use any browser - not just Firefox.

In your browser, you should see a screen similar to the following:



Note that this web interface is available from any machine that has access to your host machine via the network. This means that test operations and test results are available to anyone with access to your machine. You can configure Jenkins with different security to avoid this.

7.5.8 Access the Fuego docker command line

For some Fuego operations, it is handy to use the command line (shell prompt) inside the Docker container. In particular, parts of the remaining setup of your Fuego system involve running the `ftc` command line tool.

Some `ftc` commands can be run outside the container, but other require that you execute the command inside the container.

To start another shell prompt inside the currently running Fuego docker container, you can use the script `fuegosh`. This helper script is located at: `fuego/fuego-ro/scripts/fuegosh`. You may find it convenient to copy this script to a `bin` directory on your system (either `/usr/local/bin` or `~/bin`) that is already in your `PATH`.

If you run `fuegosh`, it will place you at a shell prompt inside the Fuego docker container, like so:

```
$ fuegosh
root@hostname:/#
```

7.5.9 Remaining steps

Fuego is now installed and ready for test operations. However, some steps remain in order to use it with your hardware. You need to:

- Add one or more hardware boards (board definition files)
- Add a toolchain
- Populate the Jenkins interface with test jobs

These steps are described in subsequent sections of this documentation.

See:

- [*Adding a Board*](#)
- [*Adding a Toolchain*](#)
- [*Adding Test Jobs to Jenkins*](#)

7.5.10 Alternative installation configurations

The default installation of Fuego installs the entire Fuego system, including Jenkins and the Fuego core, into a Docker container running on a host system, which Jenkins running on port 8090. However, it is possible to install Fuego in other configurations.

The configuration alternatives that are supported are:

- Install using a different TCP/IP port for Jenkins
- Install without the Jenkins server
- Install directly to your host (not inside a container)

With a different Jenkins TCP/IP port

By default the Fuego uses TCP/IP port 8090, but this can be changed to another port. This can be used to avoid a conflict with a service already using port 8090 on your host machine, or so that multiple instances of Fuego can be run simultaneously.

To use a different port than 8090 for Jenkins, specify it after the image name on the command line when you run `install.sh`. Note that this means that you must specify a Docker image name in order to specify a non-default port. For example:

```
$ ./install.sh fuego 7777
```

This would install Fuego, with an Docker image name of `fuego`, a Docker container name of `fuego-container`, and with Jenkins configured to run on port 7777

Without Jenkins

Some Fuego users have their own front-ends or back-ends, and don't need to use the Jenkins CI server to control Fuego tests, or visualize Fuego test results. `install.sh` supports the option `-nojenkins` which produces a Docker container without the Jenkins server. This reduces the overhead of the Docker container by quite a bit, for those users.

Inside the Docker container, the Fuego core is still available. Boards, toolchains, and tests are configured normally, but the `ftc` command line tool is used to execute tests. There is no need to use any of the `ftc` functions to manage nodes, jobs or views in the Jenkins system. `ftc` is used to directly execute tests using `ftc run-test`, and results can be queried using `ftc list-runs` and `ftc gen-report`.

When using Fuego with a different results visualization backend, the user will use `ftc put-run` to send the test result data to the configured back end.

Without a container

Usually, for security and test reproducibility reasons, Fuego is executed inside a Docker container on your host machine. That is, the default installation of Fuego will create a Docker container using all the software that is needed for Fuego's tests. However, in some configurations it is desirable to execute Fuego directly on a host machine (not inside a Docker container). A user may have a dedicated machine, or they may want to avoid the overhead of running a Docker container.

A separate install script, called `install-debian.sh` can be used in place of `install.sh` to install the Fuego system onto a Debian-based Linux distribution.

Please note that installing without a container is not advised unless you know exactly what you are doing. In this configuration, Fuego will not be able to manage host-side test dependencies for you correctly.

Please note also that executing without a container presents a possible security risk for your host. Fuego tests can run arbitrary bash instruction sequences as part of their execution. So there is a danger when running tests from unknown third parties that they will execute something on your test host that breaches the security, or that inadvertently damages you filesystem or data.

However, despite these drawbacks, there are test scenarios (such as installing Fuego directly to a target board), where this configuration makes sense.

7.6 Adding a Board

7.6.1 Overview

To add your own board to Fuego, there are five main steps:

1. Make sure you can access the target via ssh, serial or some other connection
2. Decide whether to use an existing user account, or to create a user account specifically for testing
3. Create a test directory on the target
4. Create a board file (on the host)
5. Add your board as a node in the Jenkins interface

1 - Set up communication to the target board

In order for Fuego to test a board, it needs to communicate with it from the host machine where Fuego is running.

The most common way to do this is to use ‘ssh’ access over a network connection. The target board needs to run an ssh server, and the host machine connects to it using the ‘ssh’ client.

The method of setting an ssh server up on a board varies from system to system, but sample instructions for setting up an ssh server on a raspberry pi are located here: *Raspberry Pi Fuego Setup*

Another method that can work is to use a serial connection between the host and the board’s serial console. Setting this up is outside the scope of this current documentation, but Fuego uses the “serio” package to accomplish this. I

2 - Decide on user account for testing (creating one if needed)

On your target board, a user account is required in order to run tests.

The user account used by Fuego is determined by your board file, which you will configure manually in step 4. You need to decide which account to use. There are three options:

- use the root account
- use an existing account
- use a new account, dedicated to testing

There are pros and cons to each approach.

My personal preference is to use the root account. Several tests in Fuego require root privileges. If you are working with a test board, that you can re-install easily, using the ‘root’ account will allow you to run the greatest number of tests. However, this should not be used to test machines that are in production. A Fuego test can run all kinds of commands, and you should not trust that tests will not destroy your machine (either accidentally or via some malicious intent).

If you don’t use ‘root’, then you can either use an existing account, or create a new account. In most circumstances it is worthwhile to create a new account dedicated to testing. However, you may not have sufficient privileges on your board to do this.

In any event, at this point, decide which account you will use for testing with Fuego, and note it to include in the board file, described later.

3 - Create test directory on target

First, log in to your target board, and create a directory where tests can be run. Usually, you do this as root, and a commonly used directory for this is “/home/fuego”. To do this, do the following:

For target with network connectivity :

```
$ ssh root@your_target
<target>$ mkdir /home/fuego
<target>$ exit
```

For target with Serial connectivity :

Use minicom or any other serial terminal tool. Login to the target by giving username and password. Create the directory ‘fuego’ as below:

```
<target>$ mkdir /home/fuego
```

Create board file

Now, create your board file. The board files reside in <fuego-source-dir>/fuego-ro/boards, and each file has a filename with the name of the board, with the extension “.board”.

The easiest way to create a board file is to copy an existing one, and edit the variables to match those of your board. The following instructions are for a board called ‘myboard’, that has networking access, an ssh daemon running on target, and the ARM architecture.

Do the following:

```
$ cd fuego-ro/boards
$ cp template-dev.board myboard.board
$ vi myboard.board
```

Note: You can use your own editor in place of ‘vi’*

Set board parameters

A board file has parameters which define how Fuego interacts with your board. There are lots of different parameters, but the most important to get started quickly (and the most commonly edited) are:

TRANSPORT parameters

Each board needs to specify how Fuego will communicate with it. This is done by specifying a TRANSPORT, and a few variables associated with that transport type.

- TRANSPORT - this specifies the transport to use with the target
 - there are three transport types currently supported: ‘ssh’, ‘serial’, ‘ttc’
 - Most boards will use the ‘ssh’ or ‘serial’ transport type
 - ex: TRANSPORT=”ssh”

Most targets require the following:

- LOGIN - specifies the user account to use for Fuego operations
- PASSWORD - specifies the password for that account (if any)

There are some parameters that are specific to individual transports.

For targets using ssh:

- IPADDR
- SSH_PORT
- SSH_KEY

IPADDR is the network address of your board. SSH_PORT is the port where the ssh daemon is listening for connections. By default this is 22, but you should set this to whatever your target board uses. SSH_KEY is the absolute path where an SSH key file may be found (to allow password-less access to a target machine).

An example would be:

- SSH_KEY="/fuego-ro/boards/myboard_id_rsa"

SSH_PORT and SSH_KEY are optional.

For targets using serial:

- SERIAL
- BAUD
- IO_TIME_SERIAL

SERIAL is serial port name used to access the target from the host. This is the name of the serial device node on the host (or in the container).this is specified without the /dev/ prefix.

Some examples are:

- ttyACM0
- ttyACM1
- ttyUSB0

BAUD is the baud-rate used for the serial communication, for eg. "115200".

IO_TIME_SERIAL is the time required to catch the command's response from the target. This is specified as a decimal fraction of a second, and is usually very short. A time that usually works is "0.1" seconds.

- ex: IO_TIME_SERIAL="0.1"

This value directly impacts the speed of operations over the serial port, so it should be adjusted with caution. However, if you find that some operations are not working over the serial port, try increasing this value (in small increments - 0.15, 0.2, etc.)

Note: In the case of TRANSPORT="serial", Please make sure that docker container and Fuego have sufficient permissions to access the specified serial port. You may need to modify docker-create-usb-privileged-container.sh prior to making your docker image, in order to make sure the container can access the ports.

Also, if check that the host filesystem permissions on the device node (e.g /dev/ttyACM0 allows access. From inside the container you can try using the sersh or sercp commands directly, to test access to the target.

For targets using ttc:

- TTC_TARGET

TTC_TARGET is the name of the target used with the 'ttc' command.

Other parameters

- BOARD_TESTDIR
- ARCHITECTURE
- TOOLCHAIN
- DISTRIB
- BOARD_CONTROL

The BOARD_TESTDIR directory is an absolute path in the filesystem on the target board where the Fuego tests are run. Normally this is set to something like “/home/fuego”, but you can set it to anything. The user you specify for LOGIN should have access rights to this directory.

The ARCHITECTURE is a string describing the architecture used by toolchains to build the tests for the target.

The TOOLCHAIN variable indicates the toolchain to use to build the tests for the target. If you are using an ARM target, set this to “debian-armhf”. This is a default ARM toolchain installed in the docker container, and should work for most ARM boards.

If you are not using ARM, or for some reason the pre-installed arm toolchains don’t work for the Linux distribution installed on your board, then you will need to install your own SDK or toolchain. In this case, follow the steps in [[Adding a toolchain]], then come back to this step and set the TOOLCHAIN variable to the name you used for that operation.

For other variables in the board file, see the section below.

The DISTRIB variable specifies attributes of the Linux distribution running on the board, that are used by Fuego. Currently, this is mainly used to tell Fuego what kind of system logger the operating system on the board has. Here are some options that are available:

- base.dist - a “standard” distribution that implements syslogd-style system logging. It should have the commands: logread, logger, and /var/log/messages
- nologread.dist - a distribution that has no ‘logread’ command, but does have /var/log/messages
- nosyslogd.dist - a distribution that does not have syslogd-style system logging.

If DISTRIB is not specified, Fuego will default to using “nosyslogd.dist”.

The BOARD_CONTROL variable specifies the name of the system used to control board hardware operations. When Fuego is used in conjunction with board control hardware, it can automate more testing functionality. Specifically, it can reboot the board, or re-provision the board, as needed for testing. As of the 1.3 release, Fuego only supports the ‘tte’ board control system. Other board control systems will be introduced and supported over time.

Add node to Jenkins interface

Finally, add the board in the Jenkins interface.

In the Jenkins interface, boards are referred to as “Nodes”.

You can see a list of the boards that Fuego knows about using:

- \$ ftc list-boards

When you run this command, you should see the name of the board you just created.

You can see the nodes that have already been installed in Jenkins with:

- \$ ftc list-nodes

To actually add the board as a node in jenkins, inside the docker container, run the following command at a shell prompt:

- `$ ftc add-nodes -b <board_name>`

7.6.2 Board-specific test variables

The following other variables can also be defined in the board file:

- `MAX_REBOOT_RETRIES`
- `FUEGO_TARGET_TMP`
- `FUEGO_BUILD_FLAGS`

See [Variables](#) for the definition and usage of these variables.

General Variables

File System test variables (SATA, USB, MMC)

If running filesystem tests, you will want to declare the Linux device name and mountpoint path, for the filesystems to be tested. There are three different device/mountpoint options available depending on the testplan you select (SATA, USB, or MMC). Your board may have all of these types of storage available, or only one.

To prepare to run a test on a filesystem on a sata device, define the SATA device and mountpoint variables for your board.

For example, if you had a SATA device with a mountable filesystem accessible on device `/dev/sdb1`, and you have a directory on your target of `/mnt/sata` that can be used to mount this device at, you could declare the following variables in your board file.

- `SATA_DEV="/dev/sdb1"`
- `SATA_MP="/mnt/sata"`

You can define variables with similar names (`USB_DEV` and `USB_MP`, or `MMC_DEV` and `MMC_MP`) for USB-based filesystems or MMC-based filesystems.

LTP test variables

LTP (the Linux Test Project) test suite is a large collection of tests that require some specialized handling, due to the complexity and diversity of the suite. LTP has a large number of tests, some of which may not work correctly on your board. Some of the LTP tests depend on the kernel configuration or on aspects of your Linux distribution or your configuration.

You can control whether the LTP posix test succeeds by indicating the number of positive and negative results you expect for your board. These numbers are indicated in test variables in the board file:

- `LTP_OPEN_POSIX_SUBTEST_COUNT_POS`
- `LTP_OPEN_POSIX_SUBTEST_COUNT_NEG`

You should run the LTP test yourself once, to see what your baseline values should be, then set these to the correct values for your board (configuration and setup).

Then, Fuego will report any deviation from your accepted numbers, for LTP tests on your board.

LTP may also use these other test variables defined in the board file:

- `FUNCTIONAL_LTP_HOMEDIR` - If this variable is set, it indicates where a pre-installed version of LTP resides in the board's filesystem. This can be used to avoid a lengthy deploy phase on each execution of LTP.

- `FUNCTIONAL_LTP_BOARD_SKIPLIST` - This variable has a list of individual LTP test programs to skip.

See `Functional.LTP` for more information about the LTP test, and test variables used by it.

7.7 Adding a toolchain

7.7.1 Introduction

In order to build tests for your target board, you need to install a toolchain (often in the form of an SDK) into the Fuego system, and let Fuego know how to access it.

Adding a toolchain to Fuego consists of these steps:

1. obtain (generate or retrieve) the toolchain
2. copy the toolchain to the container
3. install the toolchain inside the container
4. create a `-tools.sh` file for the toolchain
5. reference the toolchain in the appropriate board file

7.7.2 Obtain a toolchain

First, you need to obtain a toolchain that will work with your board. You should have a toolchain that produces software which will work with the Linux distribution on your board. This is usually obtained from your build tool, if you are building the distribution yourself, or from your semiconductor supplier or embedded Linux OS vendor, if you have been provided the Linux distribution from an external source.

Installing a Debian cross-toolchain target

If you are using an Debian-based target, then to get started, you may use a script to install a cross-compiler toolchain into the container. For example, for an ARM target, you might want to install the Debian `armv7hf` toolchain. You can even try a Debian toolchain with other Linux distributions. However, if you are not using Debian on your target board, there is no guarantee that this will produce correct software for your board. It is much better to install your own SDK for your board into the fuego system.

To install a Debian cross toolchain into the container, get to the shell prompt in the container and use the following script:

- `/fuego-ro/toolchains/install_cross_toolchain.sh`

To use the script, pass it the argument naming the cross-compile architecture you are using. Available values are:

- `arm64 armel armhf mips mipsel powerpc ppc64el`

Execute the script, inside the docker container, with a single command-line option to indicate the cross-toolchain to install. You can use the script more than once, if you wish to install multiple toolchains.

Example:

- `# /fuego-ro/toolchains/install_cross_toolchain.sh armhf`

The Debian packages for the specified toolchain will be installed into the docker container.

Building a Yocto Project SDK

When you build an image in the Yocto Project, you can also build an SDK to go with that image using the ‘-c do_populate_sdk’ build step with bitbake.

To build the SDK in Yocto Project, inside your yocto build directory do:

- bitbake <image-name> -c do_populate_sdk

This will build an SDK archive (containing the toolchain, header files and libraries needed for creating software on your target, and put it into the directory <build-root>/tmp/deploy/sdk/

For example, if you are building the ‘core-image-minimal’ image, you would execute:

```
$ bitbake core-image-minimal -c do_populate_sdk
```

At this step look in tmp/deploy/sdk and note the name of the sdk install package (the file ending with .sh).

7.7.3 Install the SDK in the docker container

To allow fuego to use the SDK, you need to install it into the fuego docker container. First, transfer the SDK into the container using docker cp.

With the container running, on the host machine do:

- docker ps (note the container id)
- docker cp tmp/deploy/sdk/<sdk-install-package> <container-id>:/tmp

This last command will place the SDK install package into the /tmp directory in the container.

Now, install the SDK into the container, wherever you would like. Many toolchains install themselves under /opt.

At the shell inside the container, run the SDK install script (which is a self-extracting archive):

- /tmp/poky-...sh
 - during the installation, select a toolchain installation location, like: /opt/poky/2.0.1

These instructions are for an SDK built by the Yocto Project. Similar instructions would apply for installing a different toolchain or SDK. That is, get the SDK into the container, then install it inside the container.

7.7.4 Create a -tools.sh file for the toolchain

Now, fuego needs to be told how to interact with the toolchain. During test execution, the fuego system determines what toolchain to use based on the value of the TOOLCHAIN variable in the board file for the target under test. The TOOLCHAIN variable is a string that is used to select the appropriate ‘<TOOLCHAIN>-tools.sh’ file in /fuego-ro/toolchains.

You need to determine a name for this TOOLCHAIN, and then create a file with that name, called \$TOOLCHAIN-tools.sh. So, for example if you created an SDK with poky for the qemuarm image, you might call the TOOLCHAIN “poky-qemuarm”. You would create a file called “poky-qemuarm-tools.sh”

The -tools.sh file is used by Fuego to define the environment variables needed to interact with the SDK. This includes things like CC, AR, and LD. The complete list of variables that this script needs to provide are described on the page [[tools.sh]]

Inside the `-tools.sh` file, you execute instructions that will set the environment variables needed to build software with that SDK. For an SDK built by the Yocto Project, this involves setting a few variables, and calling the `environment-setup...` script that comes with the SDK. For SDKs from other sources, you can define the needed variables by directly exporting them.

Here is an example of the `tools.sh` script for `poky-qemuarm`. This is in the sample file `/fuego-ro/toolchains/poky-qemuarm-tools.sh`:

```
# fuego toolchain script
# this sets up the environment needed for fuego to use a
# toolchain
# this includes the following variables:
# CC, CXX, CPP, CXXCPP, CONFIGURE_FLAGS, AS, LD, ARCH
# CROSS_COMPILE, PREFIX, HOST, SDKROOT
# CFLAGS and LDFLAGS are optional
#
# this script is sourced by /fuego-ro/toolchains/tools.sh

POKY_SDK_ROOT=/opt/poky/2.0.1
export SDKROOT=${POKY_SDK_ROOT}/sysroots/
armv5e-poky-linux-gnueabi

# the Yocto project environment setup script changes PATH so
# that python uses
# libs from sysroot, which is not what we want, so save the
# original path
# and use it later
ORIG_PATH=$PATH

PREFIX=arm-poky-linux-gnueabi
source ${POKY_SDK_ROOT}/environment-setup-armv5e-
poky-linux-gnueabi

HOST=arm-poky-linux-gnueabi

# don't use PYTHONHOME from environment setup script
unset PYTHONHOME
env -u PYTHONHOME
```

7.7.5 Reference the toolchain in a board file

Now, to use that SDK for building test software for a particular target board, set the value of the `TOOLCHAIN` variable in the board file for that target.

Edit the board file:

- `vi /fuego-ro/boards/myboard.board`

And add (or edit) the line:

- `TOOLCHAIN="poky-qemuarm"`

7.7.6 Notes

Python execution

You may notice that some of the example scripts set the environment variable `ORIG_PATH`. This is used by the function `[[function_run_python|run_python]]` internally to execute the container's default python interpreter, instead of the interpreter that was built by the Yocto Project.

7.8 Adding test jobs to Jenkins

Before performing any tests with Fuego, you first need to add Jenkins jobs for those tests in Jenkins.

To add jobs to Jenkins, you use the 'ftc' command line tool.

Fuego comes with over a hundred different tests, and not all of them will be useful for your environment or testing needs.

In order to add jobs to Jenkins, you first need to have created a Jenkins node for the board for which you wish to add the test. If you have not already added a board definition, or added your board to Jenkins, please see: [Adding a board](#)

Once your board is defined as a Jenkins node, you can add test jobs for it.

There are two ways of adding test jobs, individually, and using testplans. In both cases, you use the 'ftc add-jobs' command.

7.8.1 Selecting tests or plans

The list of all tests that are available can be seen by running the command 'ftc list-tests'.

Run this command inside the docker container, by going to the shell prompt inside the Fuego docker container, and typing

```
(container_prompt)$ ftc list-tests
```

To see the list of plans that come pre-configured with Fuego, use the command 'ftc list-plans'.

```
(container_prompt)$ ftc list-plans
```

A plan lists a set of tests to execute. You can examine the list of tests that a testplan includes, by examining the testplan file. The testplan files are in JSON format, and are in the directory `fuego-core/engine/overlays/testplans`.

7.8.2 Adding individual tests

To add an individual test, add it using the 'ftc add-jobs' command. For example, to add the test "Functional.hello_world" for the board "beaglebone", you would use the following command:

```
(container prompt)$ ftc add-job -b beaglebone -t  
Functional.hello_world
```


Configuring job options

When Fuego executes a test job, several options are available to control aspects of job execution. These can be configured on the ‘ftc add-job’ command line.

The options available are:

- timeout
- rebuild flag
- reboot flag
- precleanup flag
- postcleanup flag

See ‘ftc add-jobs help’ for details about these options and how to specify them.

Adding tests for more than one board

If you want to add tests for more than one board at a time, you can do so by specifying multiple board names after the ‘-b’ option with ‘ftc add-jobs’. Board names should be a single string argument, with individual board names separated by commas.

For example, the following would add a job for Functional.hello_world to each of the boards rpi1, rpi2 and beaglebone.

```
(container prompt)$ ftc add-job -b rpi1,rpi2,beaglebone -t  
Functional.hello_world
```

7.8.3 Adding jobs based on testplans

A testplan is a list of Fuego tests with some options for each one. You can see the list of testplans in your system with the following command:

```
(container prompt)$ ftc list-plans
```

To create a set of jobs related to docker image testing, for the ‘docker’ board on the system, do the following:

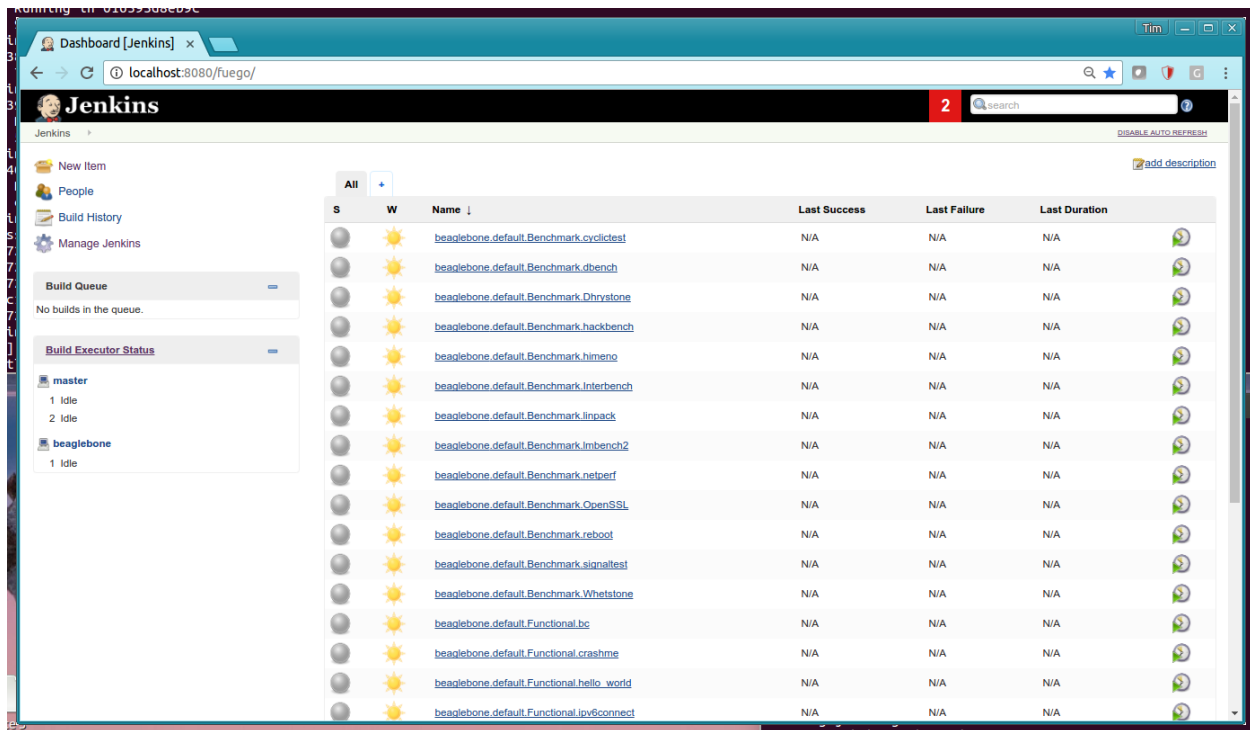
```
(container prompt)$ ftc add-jobs -b docker -p testplan_docker
```

To create a set of jobs for a board called ‘beaglebone’, do the following:

```
(container prompt)$ ftc add-jobs -b myboard -p testplan_smoketest
```

The “smoketest” testplan has about 20 tests that exercise a variety of features on a Linux system. After running these commands, a set of jobs will appear in the Jenkins interface.

Once this is done, your Jenkins interface should look something like this:



7.9 Adding views to Jenkins

It is useful to organize your Jenkins test jobs into “views”. These appear as tabs in the main Jenkins interface. Jenkins always provides a tab that lists all of the installed jobs, call “All”. Other views that you create will appear on tabs next to this, on the main Jenkins page.

You can define new Jenkins views using the Jenkins interface, but Fuego provides a command that allows you to easily create views for boards, or for sets of related tests (by name and wildcard), from the Linux command line (inside the container).

The usage line for this command is:

```
Usage: ftc add-view <view-name> [<job_spec>]
```

The view-name parameter indicates the name of the view in Jenkins, and the job-spec parameter is used to select the jobs which appear in that view.

If the job_spec is provided and starts with an ‘=’, then it is interpreted as one or more specific job names. Otherwise, the view is created using a regular expression statement that Jenkins uses to select the jobs to include in the view.

7.9.1 Adding a board view

By convention, most Fuego users populate their Jenkins interface with a view for each board in their system (well, for labs with a small number of boards, anyway).

The simplest way to add a view for a board is to just specify the board name, like so:

```
(container_prompt)$ ftc add-view myboard
```

When no job specification is provided, the ‘add-view’ command will create one by prefixing the view name with wildcards. For the example above, the job spec would consist of the regular expression “.*myboard.*”.

Customizing regular expressions

Note that if your board name is not unique enough, or is a string contained in some tests, then you might see some test jobs listed that were not specific to that board. For example, if you had a board name “Bench”, then a view you created with the view-name of “Bench”, would also include Benchmarks. You can work around this by specifying a more details regular expression for your job spec.

For example:

```
(container_prompt)$ ftc add-view Bench "Bench.*"
```

This would only include the jobs that started with “Bench” in the “Bench” view. Benchmark jobs for other boards would not be included, since they only have “Benchmark” somewhere in the middle of their job name - not at the beginning.

7.9.2 Add view by test name regular expression

This command would create a view to show LTP results for multiple boards:

```
(container_prompt)$ ftc add-view LTP
```

This example creates a view for “fuego” tests. This view would include any job that has the word “fuego” as part of it. By convention, all Fuego self-tests have part of their name prefixed with “fuego_”.

```
(container_prompt)$ ftc add-view fuego ".*fuego_.*"
```

And the following command will show all the batch jobs defined in the system:

```
(container_prompt)$ ftc add-view .*batch
```

7.9.3 Add specific jobs

If the job specification starts with “=”, it is a comma-separated list of job names. The job names must be complete, including the board name, spec name and full test name.

```
(container_prompt)$ ftc add-view network-tests =docker.default.  
Functional.ipv6connect,docker.default.Functional.netperf
```

In this command, the view would be named “network-tests”, and it would show the jobs “docker.default.Functional.ipv6connect” and “docker.default.Functional.netperf”.

7.10 Test variables

7.10.1 Introduction

When Fuego executes a test, shell environment variables are used to provide information about the test environment, test execution parameters, communications methods and parameters, and other items.

These pieces of information are originate from numerous different places. An initial set of test variables comes in the shell environment from either Jenkins or from the shell in which `ftc` is executed (depending on which one is used to invoke the test).

The information about the board being tested comes primarily from two sources:

- The board file
- The stored board variables file

Additional information comes from the testplan and test spec that are used for this particular test run. Finally, test variables can be defined on the `ftc` command line. These test variables (know as *dynamic variables*, override variables that come from other sources.

Test variables can be simple strings, or they may be shell functions.

When a test is run, Fuego gathers information from all these sources, and makes them available to the test (and uses them itself) to control test execution.

7.10.2 Board file

The board file contains static information about a board. It is processed by the overlay system, and the values inside it appear as variables in the environment of a test, during test execution.

The board file resides in `/fuego-ro/boards` and the filename ends in the string `".board"`:

- `/fuego-ro/boards/{board_name}.board`

There are a number of variables which are used by the Fuego system itself, and there may also be variables that are used by individual tests.

Common board variables

Here is a list of the foo bar variables which might be found in a board file:

- `ARCHITECTURE` - specifies the architecture of the board
- `BAUD` - baud rate for serial device (if using 'serial' transport)
- `BOARD_TESTDIR` - directory on board where tests are executed
- `BOARD_CONTROL` - the mechanism used to control board hardware (e.g. hardware reboot)
- `DISTRIB` - filename of distribution overlay file (if not the default)
- `IO_TIME_SERIAL` - serial port delay parameter (if using 'serial' transport)
- `IPADDR` - network address of the board
- `LOGIN` - specifies the user account to use for Fuego operations
- `PASSWORD` - specifies the password for the user account on the board used by Fuego
- `PLATFORM` - specifies the toolchain to use for the platform

- `SATA_DEV` - specifies a filesystem device node (on the board) for SATA filesystem tests
- `SATA_MP` - specifies a filesystem mount point (on the board) for SATA filesystem tests
- `SERIAL` - serial device on host for board's serial console (if using 'serial' transport)
- `SRV_IP` - network address of server endpoint, for networking tests (if not the same as the host)
- `SSH_KEY` - the absolute path to key file with ssh key for password-less ssh operations (e.g. `"/fuego-ro/board/myboard_id_rsa"`)
- `SSH_PORT` - network port of ssh daemon on board (if using ssh transport)
- `TRANSPORT` - this specifies the transport to use with the target
- `USB_DEV` - specifies a filesystem device node (on the board) for USB filesystem tests
- `USB_MP` - specifies a filesystem mount point (on the board) for USB filesystem tests

See [Adding a board](#) for more details about these variables.

A board may also have additional variables, including variables that are used for results evaluation for specific tests.

7.10.3 Overlay system

The overlay system gathers variables from several places, and puts them all together into a single file which is then source'd into the running test's environment.

It takes information from:

- The board files (both static and dynamic)
- The testplan
- The test spec
- The overlay files

and combines them all, using a set of priorities, into a single file called `prolog.sh`, which is then source'd into the running shell environment of the Fuego test being executed.

The overlay system is described in greater detail here: [Overlay_Generation](#)

7.10.4 Stored variables

Stored board variables are test variables that are defined on a per-board basis, and can be modified and managed under program control.

Stored variables allow the Fuego system, a test, or a user to store information that can be used by tests. This essentially creates an information cache about the board, that can be both manually and programmatically generated and managed.

The information that needs to be held for a particular board depends on the tests that are installed in the system. Thus the system needs to support ad-hoc collections of variables. Just putting everything into the static board file would not scale, as the number of tests increases.

Note: The LAVA test framework has a similar concept called a board dictionary.

One use case for this to have a "board setup" test, that scans for lots of different items, and populates the stored variables with values that are used by other tests. Some items that are useful to know about a board take time to discover (using e.g. `find` on the target board), and using a board dynamic variable can help reduce the time required to check these items.

The board stored variables are kept in the file:

- `/fuego-rw/boards/{board_name}.vars`

These variables are included in the test by the overlay generator.

Commands for interacting with stored variables

A user or a test can manipulate a board stored variable using the `ftc` command. The following commands can be used to set, query and delete variables:

- `tc query-board` - to see test variables (both regular board variables and stored variables)
- `ftc set-var` - to add or update a stored variable
- `ftc delete-var` - to delete a stored variable

`ftc query-board`

`ftc query-board` is used to view the variables associated with a Fuego board. You can use the command to see all the variables, or just a single variable.

Note that `ftc query-board` shows the variables for a test that come from both the board file and board stored variables file (that is, both ‘static’ board variables and stored variables). It does not show variables which come from testplans or spec files, as those are specific to a test.

The usage is:

- `ftc query-board <board> [-n <VARIABLE>]`

Examples:

```
$ ftc query-board myboard
$ ftc query-board myboard -n PROGRAM_BC
```

The first example would show all board variables, including functions. The second example would show only the variable `PROGRAM_BC`, if it existed, for board ‘myboard’.

`ftc set-var`

`ftc set-var` allows setting or updating the value of a board stored variable.

The usage is:

- `ftc set-var <board> <VARIABLE>=<value>`

By convention, variable names are all uppercase, and function names are lowercase, with words separated by underscores.

Example:

```
$ ftc set-var PROGRAM_BC=/usr/bin/bc
```

ftc delete-var

`ftc delete-var` removes a variable from the stored variables file.

Example:

```
$ ftc delete-var PROGRAM_BC
```

Example usage

The test `Functional.fuego_board_check` could detect the path for the `foo` binary, (e.g. `is_on_target foo PROGRAM_FOO`) and call `ftc set-var $NODE_NAME PROGRAM_FOO=$PROGRAM_FOO`. This would stay persistently defined as a test variable, so other tests could use `$PROGRAM_FOO` (with `assert_define`, or in `report` or `cmd` function calls.)

Example Stored variables

Here are some examples of variables that can be kept as stored variables, rather than static variables from the board file:

- `SATA_DEV` = Linux device node for SATA file system tests
- `SATA_MP` = Linux mount point for SATA file system tests
- `LTP_OPEN_POSIX_SUBTEST_COUNT_POS` = expected number of pass results for LTP OpenPosix test
- `LTP_OPEN_POSIX_SUBTEST_COUNT_NEG` = expected number of fail results for LTP OpenPosix test
- `PROGRAM_BC` = path to 'bc' program on the target board
- `MAX_REBOOT_RETRIES` = number of retries to use when rebooting a board

7.10.5 Spec variables

A test spec can define one or more variables to be used with a test. These are commonly used to control test variations, and are specified in a `spec.json` file.

When a spec file defines a variable associated with a named test spec, the variable is read by the overlay generator on test execution, and the variable name is prefixed with the name of the test, and converted to all upper case.

For example, support a test called `Functional.foo` had a test spec that defined the variable 'args' with a line like the following in its `spec.json` file:

```

"default": {
  "args": "-v -p2"
}

```

When the test was run with this spec (the "default" spec), then the variable `FUNCTIONAL_FOO_ARGS` would be defined, with the value `"-v -p2"`.

See `Test_Specs_and_Plans` for more information about specs and plans.

7.10.6 Dynamic variables

Another category of variables used during testing are dynamic variables. These variables are defined on the command line of `ftc run-test` using the `--dynamic-vars` option.

The purpose of these variables is to allow scripted variations when running `ftc run-test`. The scripted variables are processed and presented the same way as spec variables, which is to say that the variable name is prefixed with the test name, and converted to all upper case.

For example, if the following command was issued:

- `ftc run-test -b beaglebone -t Functional.foo --dynamic-vars *ARGS=-p*`

then during test execution the variable `FUNCTIONAL_FOO_ARGS` would be defined with the value `“-p”`.

See Dynamic Variables for more information.

7.10.7 Variable precedence

Here is the precedence of variable definition for Fuego, during test execution:

(from lowest to highest)

1. environment variable (from Jenkins or shell where ‘`ftc run-test`’ is invoked)
2. board variable (from `fuego-ro/boards/$BOARD.board` file)
3. stored variable (from `fuego-rw/boards/$BOARD.vars` file)
4. spec variable (from `spec.json` file)
5. dynamic variable (from `ftc` command line)
6. core variable (from Fuego scripts)
7. `fuego_test` variable (from `fuego_test.sh`)

Spec and dynamic variables are prefixed with the test name, and converted to upper case. That tends to keep them in a separate name space from the rest of the test variables.

7.11 Jenkins User interface

By default, Fuego uses the Jenkins continuous integration system to manage boards, tests, logs, and test results.

The Jenkins user interface is web-based. This page shows several screenshots of different pages in the Jenkins interface.

Through this interface, you can see the status of tests that have run, review the logs for tests, and schedule new tests to run on target boards. You also use this interface to add new boards and new tests to the system.

Note that Jenkins objects are:

- nodes
- jobs
- builds
- views

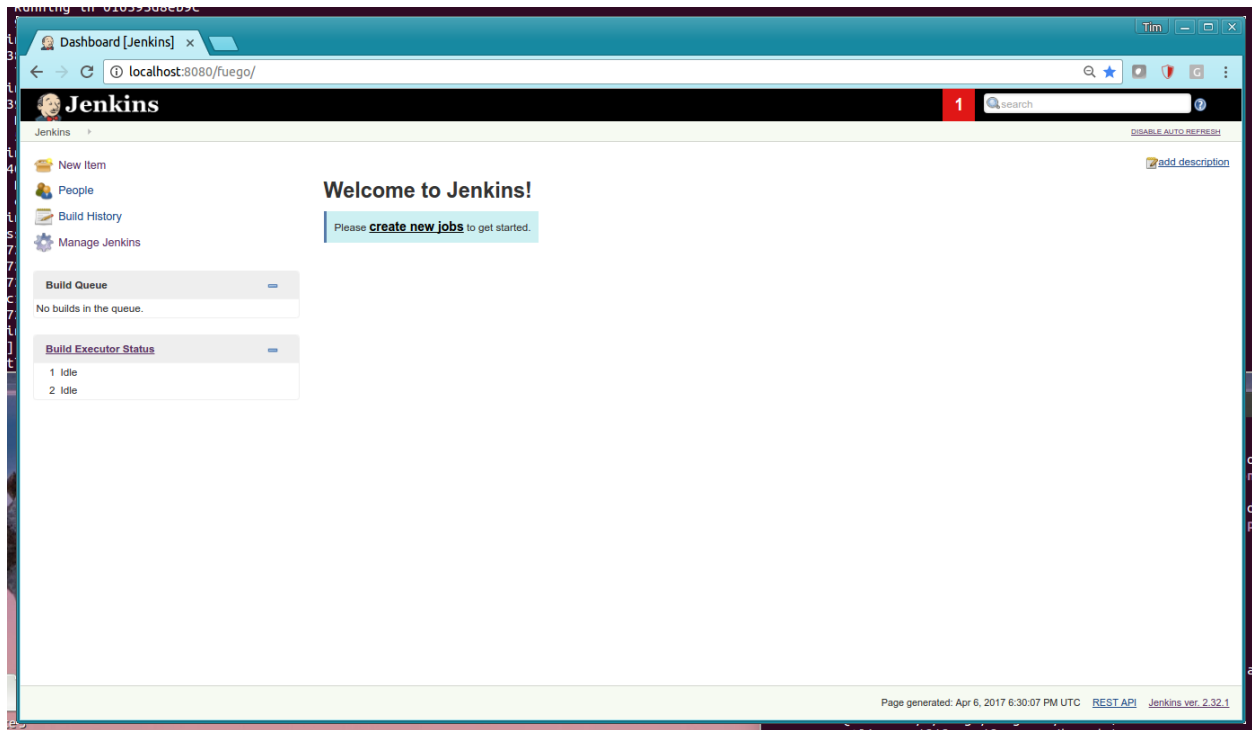
These are different from the Fuego names for the same objects. The first three of these Jenkins objects correspond to the Fuego objects of: **boards**, **tests** and **runs**, respectively.

7.11.1 Main dashboard

The main dashboard of Jenkins looks like the following:

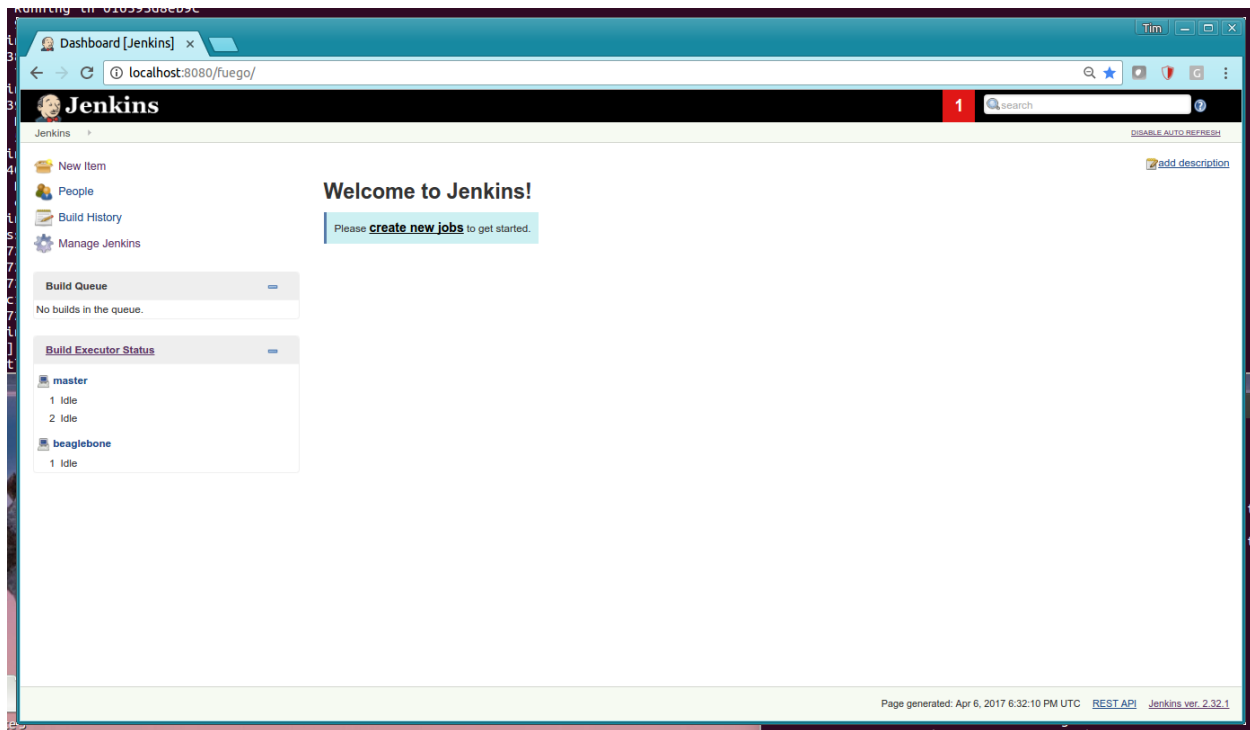
New Installation

When Fuego has just been installed, there is nothing in the Jenkins interface (no nodes, jobs or views). The interface should look something like this:



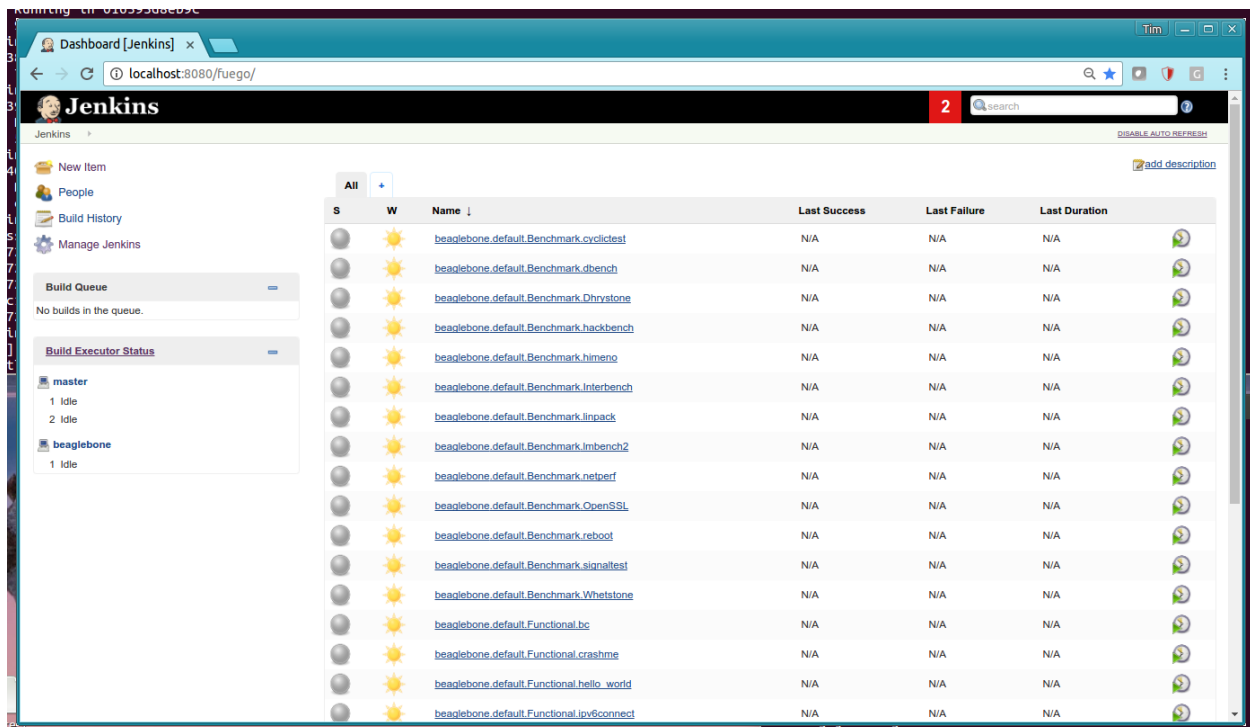
With a single node (board) added

Here is the main dashboard of Jenkins, after a single node (called 'beaglebone' in this case) has been added. Note the node (board) appears in the left sidebar under "Build Executor Status":



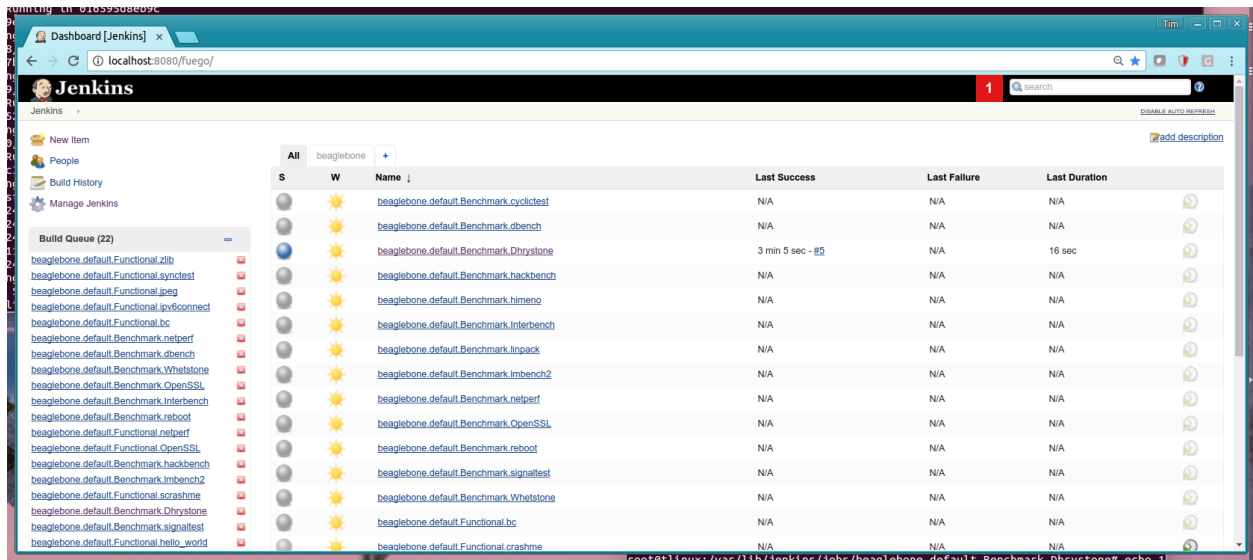
With beaglebone node and jobs

Here is the main dashboard of Jenkins, showing a single node (beaglebone) and jobs for this board.



Dashboard with jobs in Build Queue

Here is the Jenkins dashboard with a lot of jobs in the Build Queue. Note the list of jobs in the left side-bar, in the “Build Queue” pane.

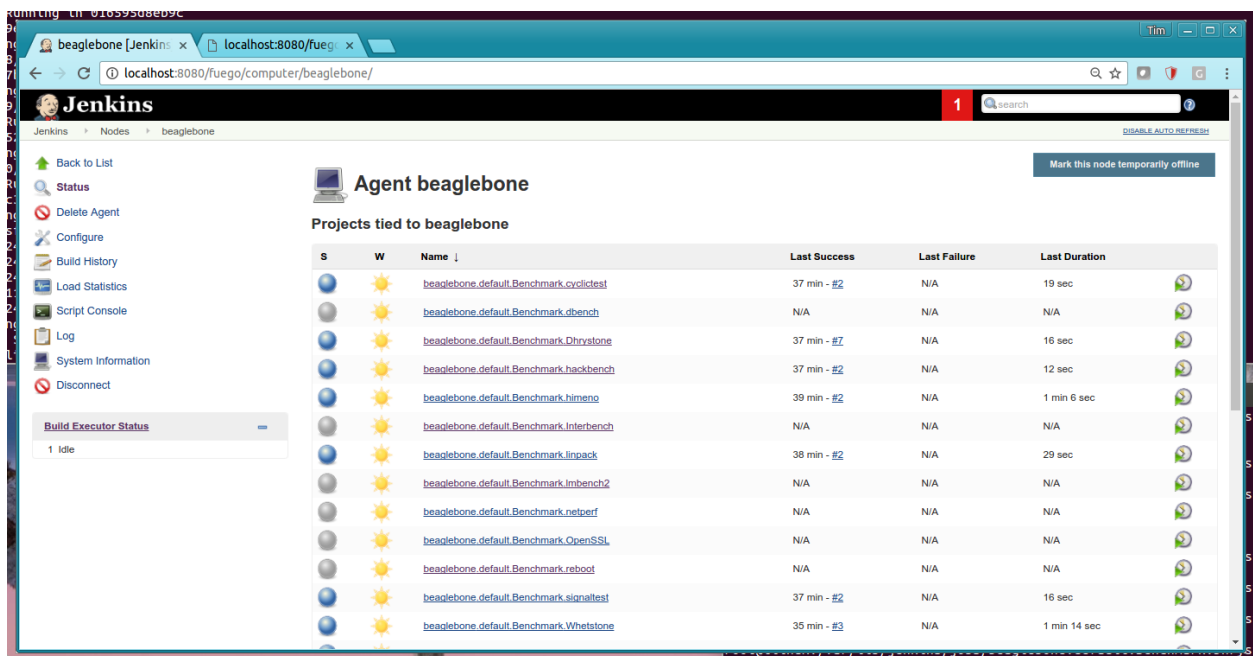


7.11.2 Node pages

If you click on the node in the **Build Executor Status** pane, then you can see a list of the jobs associated with a node.

Node status page

Here is the status for the beaglebone node.



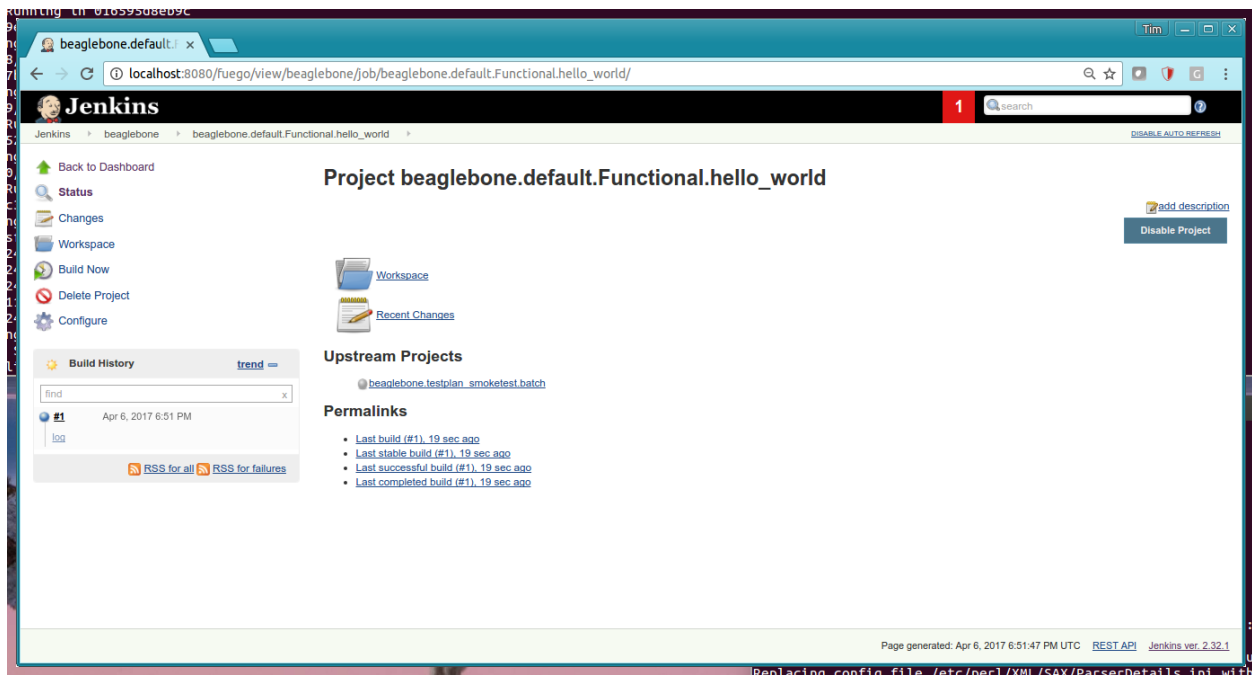
7.11.3 Job pages

If you click on a job in the Jenkins interface, you can see information about an individual job. This page shows information about the status of the job, including a Build History for the job (in the left sidebar).

You can start a job by clicking on the “Build Now” button in the left menu.

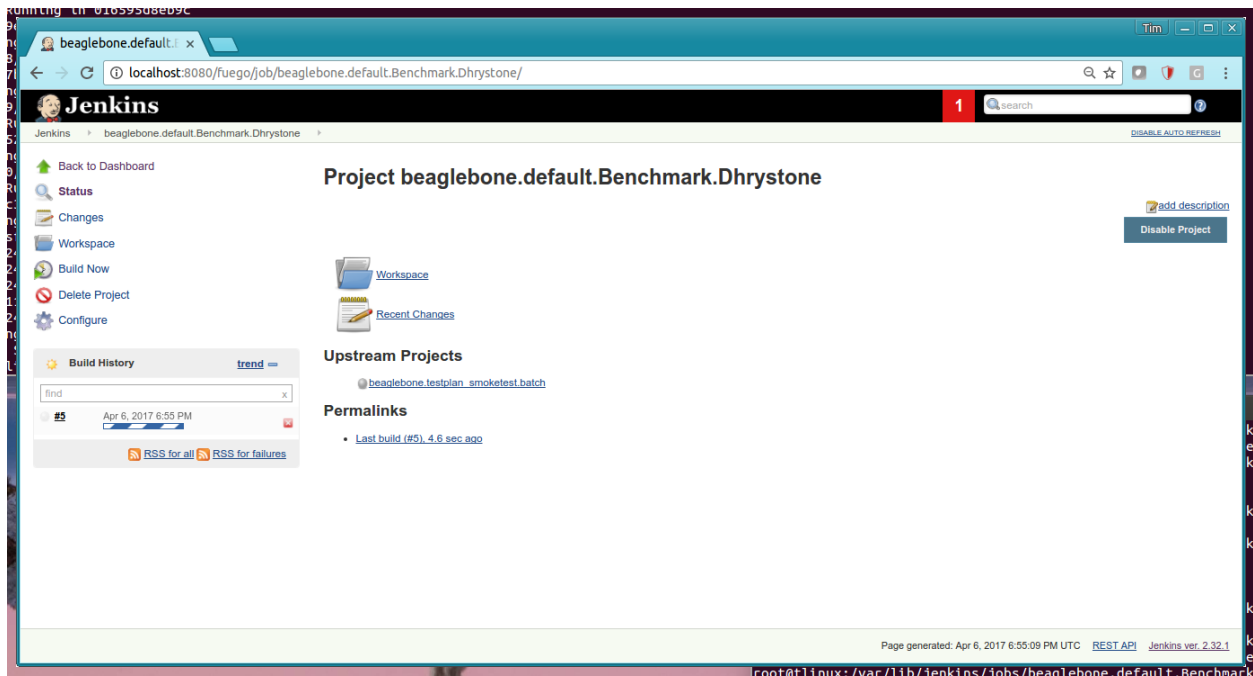
Functional job status page

Here is a page showing the status information for a Functional test called ‘hello_world’. The main area of the screen has information about the last successful and failed builds of the test. Note the left sidebar pane with “Build History”, to see individual test execution results.



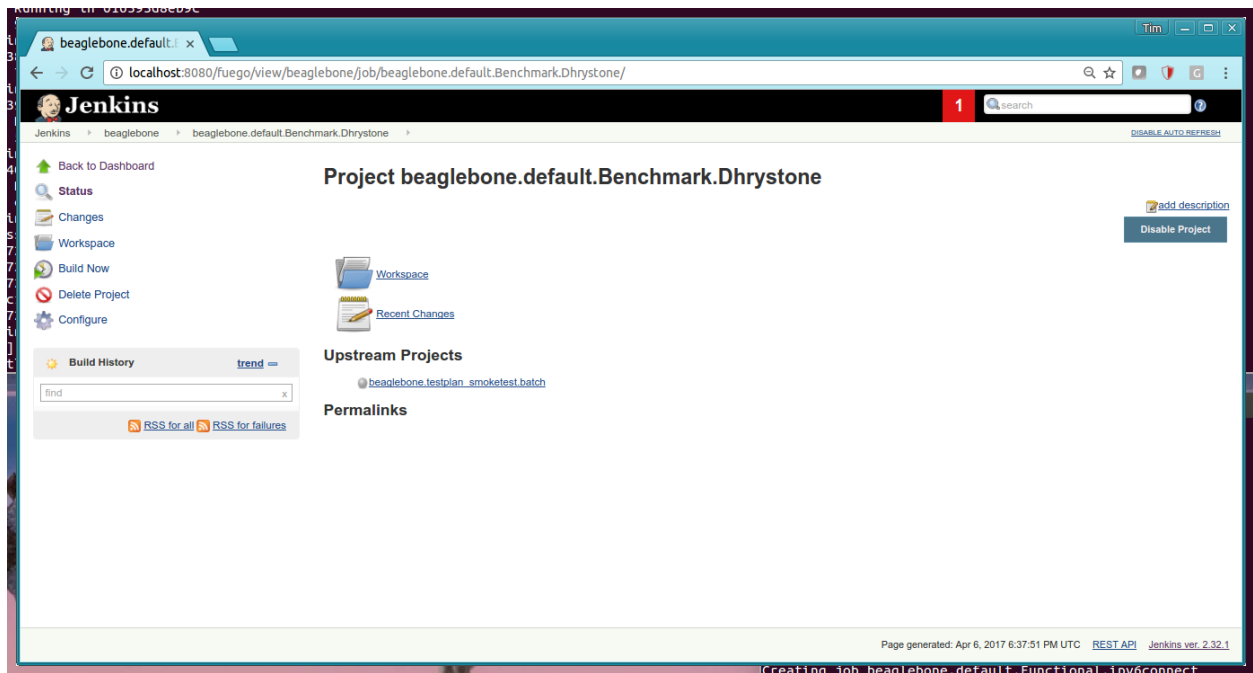
Benchmark job - starting a build

Here is a picture of a job build being started. Note the progress bar in the **Build History** pane in the left sidebar.



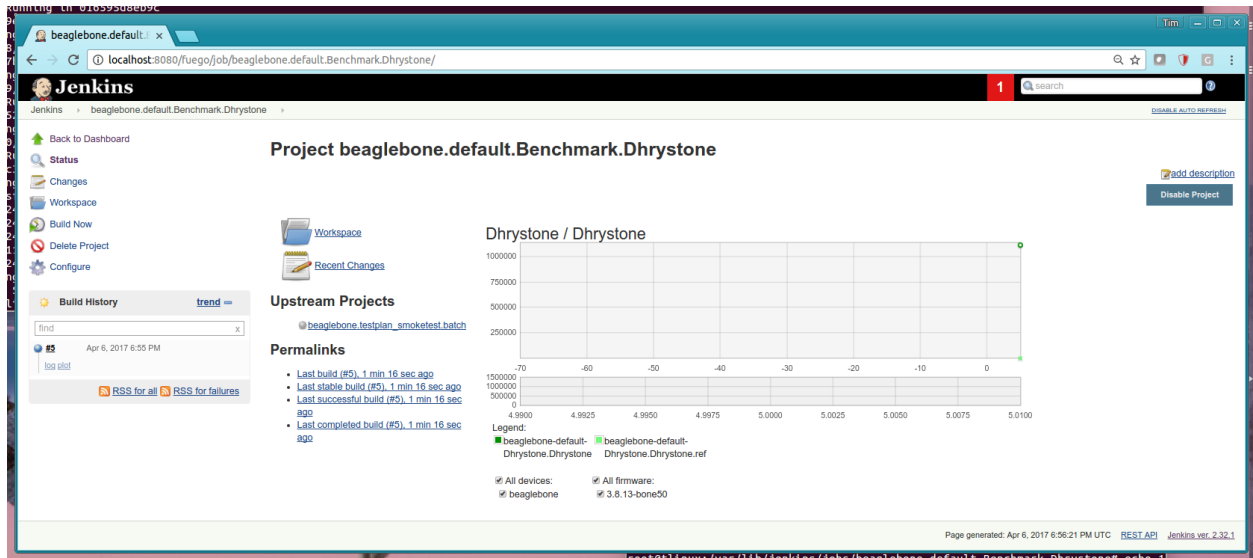
Benchmark job - before successful execution

Before a Benchmark job completes it has no data to plot on it's chart, and appears similar to a Functional Job status page:



Benchmark job - with plot of metrics

Normally, a Benchmark page shows one or more plots showing the values for data returned by this benchmark.

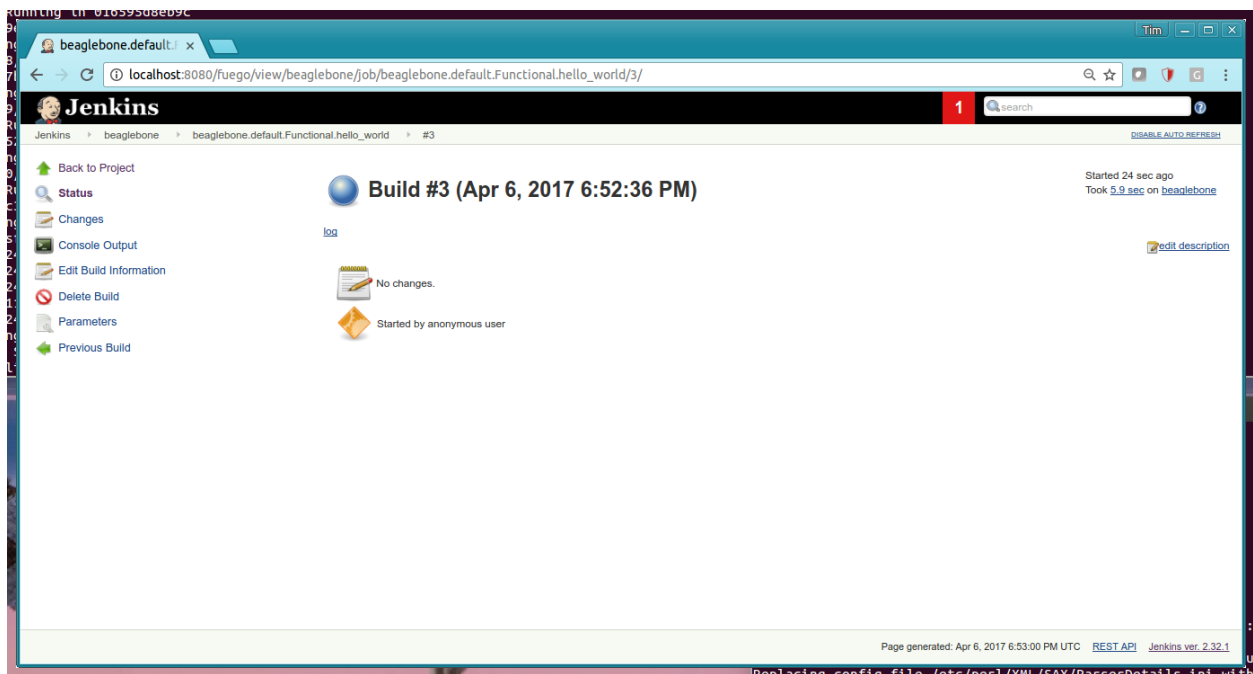


7.11.4 Build pages

A build page shows the results of a single execution of a job (test) on a board. You can click on the build number in the Jenkins interface to see this page.

Results from a job build

Here are the results from the execution of the “hello world” job. This was the results of running the Fuego test “Functional.hello_world” on a beaglebone board.



Test log results

You can examine the different logs for each test. Each test produces a log from the program that ran on the board. This is available by following a link called “log” from the ‘build’ page for that test run. You can see the console log, which shows the output of commands for this test, by clicking on “console log” in the build interface (or the build drop-down menu in the **Build History** list).

Dhrystone test log

Here are results from a run of the Dhrystone test on a beaglebone board:

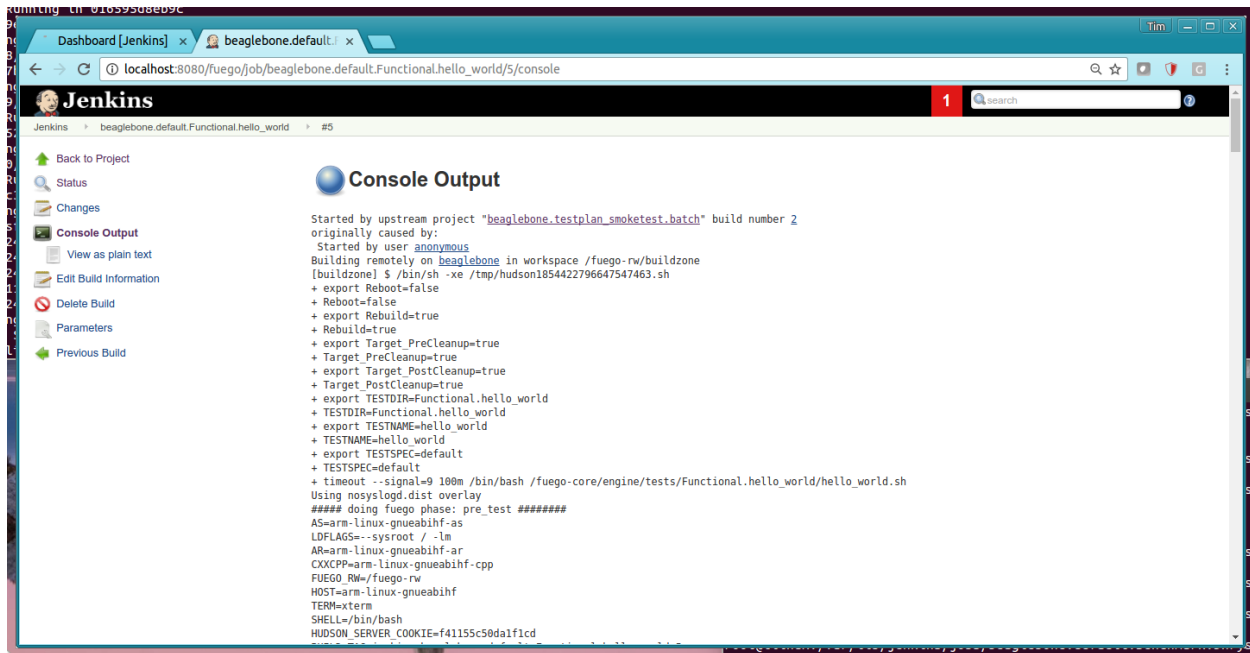
```

Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without 'register' attribute
Please give the number of runs through the benchmark:
Execution starts, 10000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:
Int_Glob: 5
  should be: 5
Bool_Glob: 1
  should be: 1
Ch_1_Glob: A
  should be: A
Ch_2_Glob: B
  should be: B
Arr_1_Glob[8]: 7
  should be: 7
Arr_2_Glob[8][7]: 10000010
  should be: Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp: 151560
  should be: (implementation-dependent)
  Discr: 0
  should be: 0
  Enum_Comp: 2
  should be: 2
  Int_Comp: 17
  should be: 17
  Str_Comp: DHRYSTONE PROGRAM, SOME STRING
  should be: DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp: 151560
  should be: (implementation-dependent), same as above
  Discr: 0
  should be: 0
  Enum_Comp: 1
  should be: 1
  Int_Comp: 18
  should be: 18
  Str_Comp: DHRYSTONE PROGRAM, SOME STRING
  should be: DHRYSTONE PROGRAM, SOME STRING
Int_1_loc: 5
  
```

Jenkins Console log

Here is the console log for a test executed on the beaglebone:



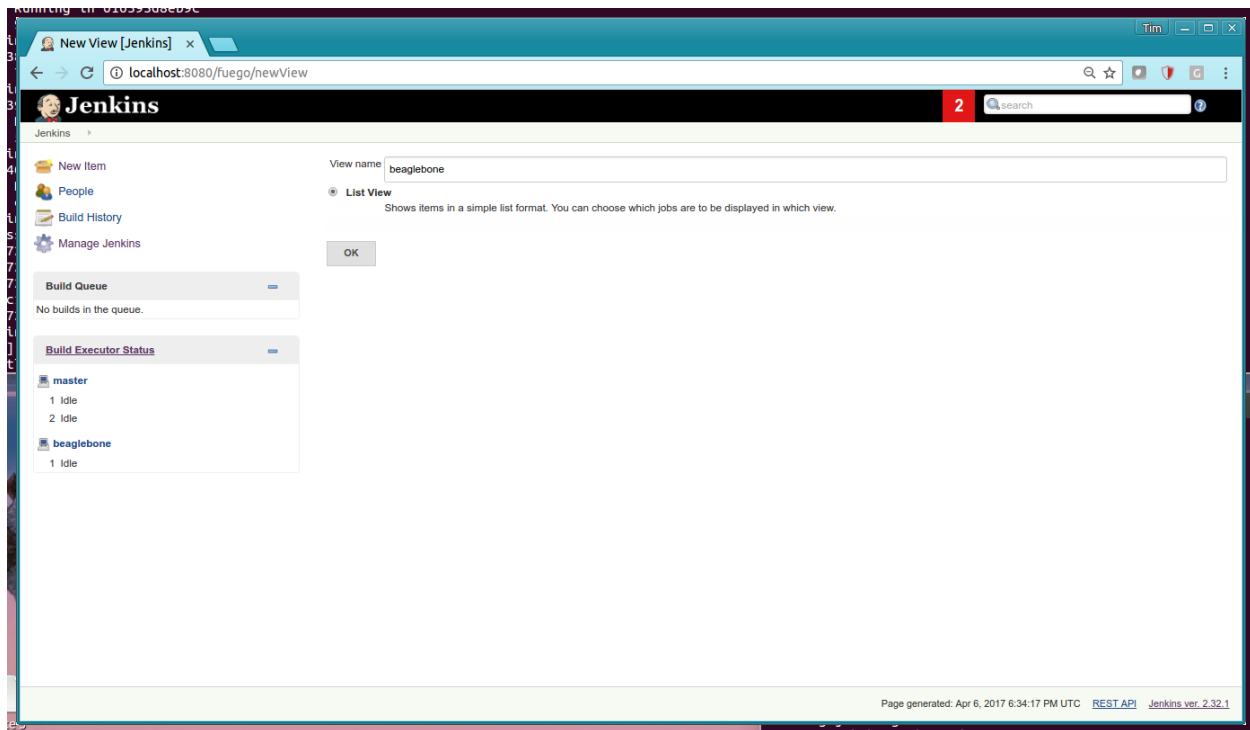
7.11.5 View pages

A **view** is an area in the main dashboard that shows a collection of jobs with a particular set of status columns for each job. They appear as tabs in the main dashboard view of Jenkins. You can create your own view to see a subset of the jobs that are available in Jenkins

Here are some screen shots showing how to add a new view to Jenkins.

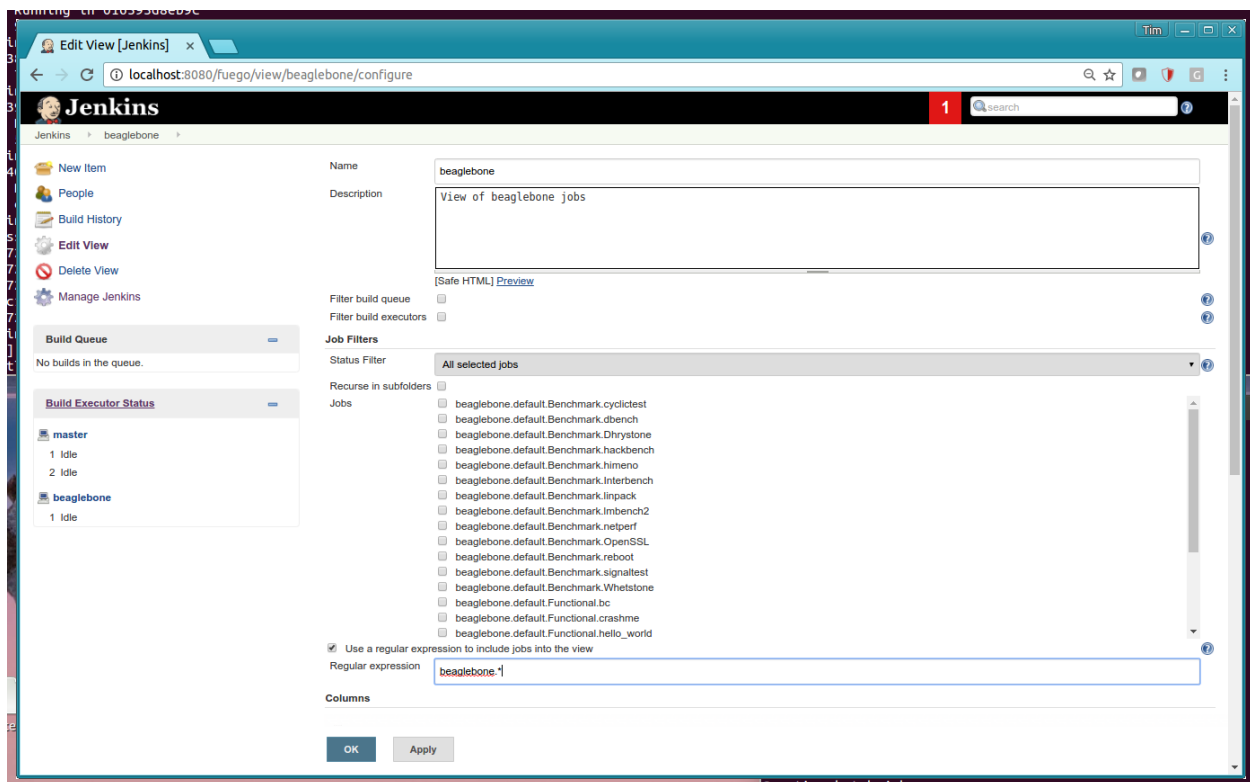
Screen to add a new view

Here is the screen to add a new view to Jenkins.



Screen to configure the view settings

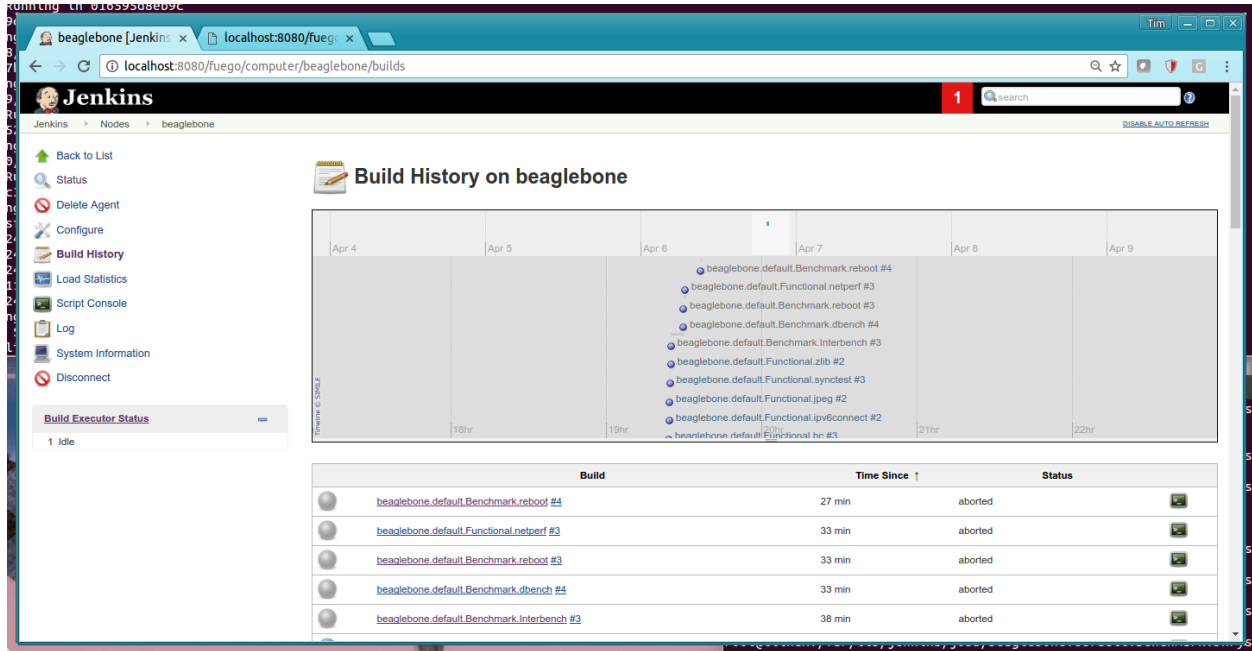
Here is the screen to configure view settings. Note the use of a regular expression to control what jobs to see in this view. You can also control what status columns to display in the view.



7.11.6 Other Jenkins pages

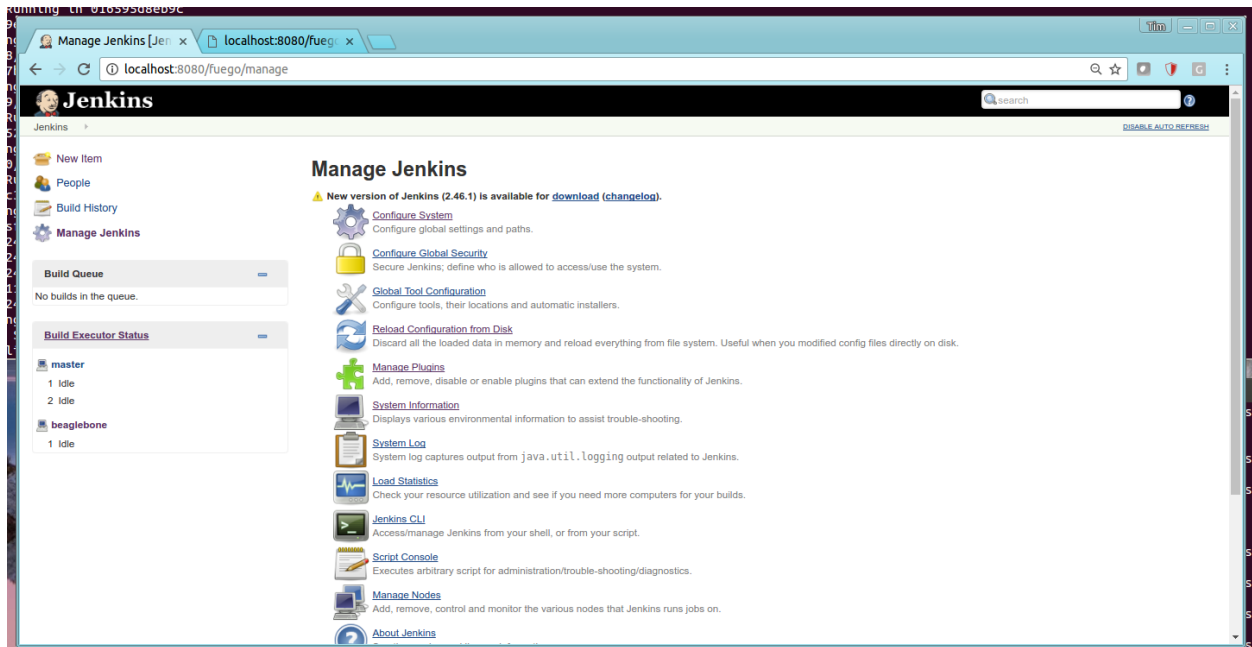
Build History

The global build history page is available by clicking on the **Build History** link in the main dashboard page of Jenkins. It shows the execution time and status for a recent time period.



Jenkins management

You can manage Jenkins using the **Manage Jenkins** page, available from the top-level dashboard page in Jenkins. From here you can update Jenkins itself, install or remove plugins, and perform other management operations for Jenkins.



7.12 Adding or Customizing a Distribution

7.12.1 Introduction

Although Fuego is configured to execute on a standard Linux distribution, Fuego supports customizing certain aspects of its interaction with the system under test. Fuego uses several features of the operating system on the board to perform aspects of its test execution. This includes things like accessing the system log, flushing file system caches, and rebooting the board. The ability to customize Fuego's interaction with the system under test is useful in case you have a non-standard Linux distribution (where, say, certain features of Linux are missing or changed), or when you are trying to use Fuego with a non-Linux system.

A developer can customize the distribution layer of Fuego in one of two ways:

- adding overlay functions to a board file by creating a new
- distribution overlay file

7.12.2 Distribution overlay file

A distribution overlay file can be added to Fuego, by adding a new “.dist” file to the directory: `fuego-core/overlays/distrib`

The *distribution* functions are defined in the file: `fuego-core/overlays/base/base-distrib.fuegoclass` These include functions for doing certain operations on your board, including:

- `ov_get_firmware`
- `ov_rootfs_reboot`
- `ov_rootfs_state`
- `ov_logger`
- `ov_rootfs_sync`
- `ov_rootfs_drop_caches`
- `ov_rootfs_oom`
- `ov_rootfs_kill`
- `ov_rootfs_logread`

You can define your own *distribution* overlay by defining a new “.dist” file in `fuego-core/overlays/distrib`. (e.g. `mydist.dist`) Basically, you inherit functions from `base-distrib.fuegoclass`, and write override functions in `mydist.dist` to perform those operations the way they need to be done on your distribution.

You can look up what each override function should do by reading the `fuegoclass` code, or looking at the function documentation at: [Test Script APIs](#)

The inheritance mechanism and syntax for Fuego overlay files is described at: [Overlay Generation](#)

The goal of the distribution abstraction layer in Fuego is to allow you to customize Fuego operations to match what is available on your target board. For example, the default (base class) `ov_rootfs_logread()` function assumes that the target board has the command “`/sbin/logread`” that can be used to read the system log. If your distribution does not have “`/sbin/logread`”, or indeed if there is no system log, then you would need to override `ov_rootfs_logread()` to do something appropriate for your distribution or OS.

Note: In fact, this is a common enough situation that there is already a ‘`nologread.dist`’ file already in the `overlay/distrib` directory.

Similarly, `ov_rootfs_kill` uses the `/proc` filesystem, `/proc/$pid/status`, and the `cat`, `grep`, `kill` and `sleep` commands on the target board to do its work. If our distribution is missing any of these, then you would need to override `ov_rootfs_kill()` with a function that did the appropriate thing on your distribution (or OS).

Existing distribution overlay files

Fuego provides a few distribution overlay files for certain situations that commonly occur in embedded Linux testing.

- `nologread.dist` - for systems that do not have a ‘logread’ command
- `nosyslogd.dist` - for systems that don’t have any system logger

7.12.3 Referencing the distribution in the board file

Inside the board file for your board, indicate the distribution overlay you are using by setting the *DISTRIB* variable.

If the *DISTRIB* variable is not set, then the default distribution overlay functions are used.

For example, if your embedded distribution of Linux does not have a system logger, you can override the normal logging interaction of Fuego by using the ‘`nosyslogd.dist`’ distribution overlay. To do this, add the following line to the board file for target board where this is the case:

```
DISTRIB="nosyslogd.dist"
```

7.12.4 Testing Fuego/distribution interactions

There is a test you can run to see if the minimal command set required by Fuego is supported on your board. It does not require a toolchain, since it only runs shell script code on the board. The test is `Functional.fuego_board_check`.

This test may work on your board, if your board supports a POSIX shell interface. However, note that this test reflects the commands that are used by Fuego core and by the default distribution overlay. If you make your own distribution overlay, you may want to create a version of this test that omits checks for things that your distribution does not support, or that adds checks for things that your distribution overlay uses to interact with the board.

7.12.5 Notes

Fuego does not yet fully support testing non-Linux operating systems. There is work-in-progress to support testing of NuttX, but that feature is not complete as of this writing. In any event, Fuego does include a ‘NuttX’ distribution overlay, which may provide some ideas if you wish to write your own overlay for a non-Linux OS.

NuttX distribution overlay

By way of illustration, here are the contents of the NuttX distribution overlay file (`fuego-core/overlays/distrib/nuttX.dist`).

```
override-func ov_get_firmware() {
    FW="$(cmd uname -a)"
}

override-func ov_rootfs_reboot() {
    cmd "reboot"
}
```

(continues on next page)

(continued from previous page)

```
override-func ov_init_dir() {
    # no-op
    true
}

override-func ov_remove_and_init_dir() {
    # no-op
    true
}

override-func ov_rootfs_state() {
    cmd "echo; date; echo; free; echo; ps; echo; mount" || \
        abort_job "Error executing rootfs_status operation on target"
}

override-func ov_logger() {
    # messages are in $@, just emit them
    echo "Fuego log messages: $@"
}

# $1 = tmp dir, $2 = before or after
override-func ov_rootfs_logread() {
    # no-op
    true
}

override-func ov_rootfs_sync() {
    # no-op
    true
}

override-func ov_rootfs_drop_caches() {
    # no-op
    true
}

override-func ov_rootfs_oom() {
    # no-op
    true
}

override-func ov_rootfs_kill() {
    # no-op
    true
}
```

Hypothetical QNX distribution

Say you wanted to add support for testing QNX with Fuego.

Here are some first steps to add a QNX distribution overlay:

- set up your board file
- create a custom QNX.dist (stubbing out or replacing base class functions as needed)
 - you could copy null.dist to QNX.dist, and deciding which items to replace with QNX-specific functionality
- add `DISTRIB="QNX.dist"` to your board file
- run the `Functional.fuego_board_check` test (using `ftc`, or adding the node and job to Jenkins and building the job using the Jenkins interface), and
- examine the console log to see what issues surface

7.13 Integration with ttc

This page describes how to use Fuego with `ttc`. `ttc` is a tool used for manual and automated access to and manipulation of target boards. It is a tool developed by Tim Bird and used at Sony for managing their board farms, and for doing kernel development on multiple different target boards at a time (including especially boards with different processors and architectures.)

This page describes how `ttc` and Fuego are integrated, so that the Fuego test framework can use `ttc` as it's transport mechanism.

You can find more information about `ttc` on the embedded Linux wiki at: http://elinux.org/Ttc_Program_Usage_Guide

7.13.1 Outline of supported functionality

Here is a rough outline of the support for `ttc` in Fuego:

- Integration for the tool and helper utilities in the container build
 - When the docker container is built, `ttc` is downloaded from Github and installed into the Docker image.
 - During this process, the path to the `ttc.conf` file is changed from `/etc/ttc.conf` to `/fuego-ro/conf/ttc.conf`
- `ttc` is a valid transport option in Fuego
 - You can specify `ttc` as the 'transport' for a board, instead of `ssh`
- `ttc` added support for `-r` as an option to the `ttc cp` command
 - This is required since Fuego uses `-r` extensively to do recursive directory copies (See `Transport_notes` for details)
- The Fuego core scripts have been modified to avoid using wildcards on `get` operations
- A new test called `Functional.fuego_transport` has been added
 - This tests use of wildcards, multiple files and directories and directory recursion with the Fuego `put` function.
 - It also indirectly tests the `get` function (the other major Fuego transport function), because logs are obtained during the test.

7.13.2 Supported operations

`ttc` has several sub-commands. Fuego currently only uses the following `ttc` sub-commands:

- `ttc run` - to run a command on the target
- `ttc cp` - to get a file from the target, and to put files to the target

Note that some other commands, such as `ttc reboot` are not used, in spite of there being similar functionality provided in fuego (see function `target reboot` and function `ov rootfs reboot`).

Finally, other commands, such as `ttc get_kernel`, `ttc get_config`, `ttc kbuild` and `ttc kinstall` are not used currently. These may be used in the future, when Fuego is expanded to have a focus on tests that require kernel rebuilding.

7.13.3 Location of `ttc.conf`

Normally, `ttc` on a host uses the default configuration file at `/etc/ttc.conf`. Fuego modifies the `ttc` installed inside the Fuego docker container, so that it uses the configuration file located at `/fuego-ro/conf/ttc.conf` as its default.

During Fuego installation, `/etc/ttc.conf` is copied to `/fuego-ro/conf` from the host machine, if it is present (and a copy of `ttc.conf` is not already there).

7.13.4 Steps to use `ttc` with a target board

Here is a list of steps to set up a target board to use `ttc`. These steps assume you have already added a board to fuego following the steps described in [Adding a board](#).

- If needed, create your Docker container using `docker-create-usb- privileged-container.sh`
 - This may be needed if you are using `ttc` with board controls that require access to USB devices (such as the Sony debug board)
 - Use the `--priv` option with `install.sh`, as documented in [Installing Fuego](#).
- Make sure that `fuego-ro/conf/ttc.conf` has the definitions required for your target board
 - Validate this by doing `ttc list` to see that the board is present, and `ttc run` and `ttc cp` commands, to test that these operations work with the board, from inside the container.
- Edit the fuego board file (found in `/fuego-ro/conf/boards /{board_name}.board`)
 - Set the `TRANSPORT` to “`ttc`”
 - Set the `TTC_TARGET` variable to the name for the target used by `ttc`
 - See the following example, for a definition for a target named ‘bbb’ (for my Beaglebone black board):

```
TRANSPORT=ttc
TTC_TARGET=bbb
```

7.13.5 modify your copy_to_cmd

In your `ttc.conf` file, you may need to make changes to the `copy_to_cmd` definitions for boards used by Fuego. Fuego allows programs to pass a `-r` argument to its internal `put` command, which in turn invokes `ttc`'s `cp` command, with the source as target and destination as the host. In other words, it ends up invoking `ttc`'s `copy_from_cmd` for the indicated target.

All instances of `copy_to_cmd` should be modified to reference a new environment variable `$copy_args`, and they should support the use of `-r` in the command arguments.

Basically, if a Fuego test uses `put -r` at any point, this needs to be supported by `ttc`. `ttc` will pass any `'-r'` seen to the subcommand in the environment variable `$copy_args`, where you can use it as needed with whatever sub-command (`cp`, `scp`, or something else) that you use to execute a `copy_to_cmd`.

See `ttc.conf.sample` and `ttc.conf.sample2` for usage examples.

7.14 Working with remote boards

Here are some general tips for working with remote boards (that is, boards in remote labs)

7.14.1 using a jump server

If you have an SSH jump server, then you can access machine directly in another lab, using the `SSH ProxyCommand` in the host settings for a board.

I found this page to be helpful: <https://www.tecmint.com/access-linux-server-using-a-jump-host/>

You should try to make each leg of the jump (from local machine to jump server, and from jump server to remote machine) password-less.

I found that if my local machine's public key was in the remote machine's authorized keys file, then I could log in without a password, even if the jump server's public key was not in the remote machine's authorized keys file.

7.14.2 Using ttc transport remotely

If you have a server that already has `ttc` configured for a bunch of board, you can accomplish a lot just by referencing `ttc` commands on that server.

For example, in your local `ttc.conf`, you can put:

```
PASSWORD=foo
USER=myuser
SSH_ARGS=-o UserKnownHostsFile=/dev/null -o StrictHostKeychecking=no -o LogLevel=QUIET

pos_cmd=ssh timdesk ttc %(target)s pos
off_cmd=ssh timdesk ttc %(target)s off
on_cmd=ssh timdesk ttc %(target)s on
reboot_cmd=ssh timdesk ttc %(target)s reboot

login_cmd=sshpass -p %(PASSWORD)s ssh %(SSH_ARGS)s -x %(USER)s@%(target)s
run_cmd=sshpass -p %(PASSWORD)s ssh %(SSH_ARGS)s -x %(USER)s@%(target)s "$COMMAND"
copy_to_cmd=sshpass -p %(PASSWORD)s scp %(SSH_ARGS)s $src %(USER)s@%(target)s:/$dest
copy_from_cmd=sshpass -p %(PASSWORD)s scp %(SSH_ARGS)s %(USER)s@%(target)s:/$src $dest
```

Note: Note that `ttc status <remote-board>` command does not work with `ttc` version 1.4.4. This is due to internal usage of `%(ip_addr)s` in the function `network_status()`, which will not be correct for the remote-board.

7.14.3 Setting up ssh ProxyCommand in the Fuego docker container

Please note that tests in Fuego are executed inside the Docker container as user 'jenkins'.

In order to set up password-less operation, or use of a jump server or ProxyCommand, you have to add appropriate items (config and keys) to `/var/lib/jenkins/.ssh`.

Please note that this may make your Docker container a security risk, as it may expose your private keys to tests. Please use caution when adding private keys or other sensitive security information to the Docker container.

7.15 Building Documentation

As of July, 2020, the Fuego documentation is currently available in 3 places:

- The `fuego-docs.pdf` generated from TEX files in the `fuego/docs/source` directory
- The FuegoTest wiki, located at: <https://fuegotest.org/wiki/Documentation>
- A set of html files in `fuego/docs/_build/html` that are generated from `.rst` files in `fuego/docs/rst_src`

The `fuego-docs.pdf` file is a legacy file that is several years old. It is only kept around for backwards compatibility. It might be worthwhile to scan it and see if any information is in it that is not in the wiki and migrate it to the wiki.

The fuegotest wiki has the currently-maintained documentation for the project. But there are several issues with this documentation:

- the wiki used is proprietary and slow
- the information is not well-organized
- the information is only available online, as separate pages
 - there is no option to build a single PDF, or use the docs offline
- there is a mixture of information in the wiki
 - not just documentation, but a crude issues tracker, random technical notes testing information, release information and other data that should not be part of official documentation

The `.rst` files are intended to be the future documentation source for the project.

7.15.1 building the outdated PDF

To build the outdated PDF, cd to `fuego/docs`, and type

```
$ make fuego-docs.pdf
```

This will use latex to build the file `fuego/docs/fuego-docs.pdf`

7.15.2 building the RST docs

The RST docs can be build in several different formats, including text, html, and pdf. You can type ‘make help’ to get a list of the possible build targets for this documentation. Output is always directed to a directory under fuego/docs/_build.

Here are some of the most popular targets:

html

```
$ make html
```

Documentation will be in fuego/docs/_build/html

The root of the documentation will be in index.html

singlehtml

```
$ make singlehtml
```

Documentation will be in fuego/docs/_build/singlehtml

The complete documentation will be in a single file: index.html (with images and other static content in fuego/docs/_build/singlehtml/_static

latexpdf

```
$ make latexpdf
```

Documentation will be in fuego/docs/_build/latexpdf/Fuego.pdf

7.16 Fuego Developer Notes

This page has some detailed notes about Fuego, Jenkins and how they interact.

7.16.1 Resources

Here are some pages in this wiki with developer information:

- *Coding style*
- *Core_interfaces*
- *Glossary*
- Fuego test results determination
- *Fuego_naming_rules*
- Fuego Object Details
- *Integration with ttc*
- *Jenkins User interface*

- Jenkins Plugins
- *License And Contribution Policy*
- Log files
- Metrics
- Overlay Generation
- ovgen feature notes
- *Parser module API*
- Test Script APIs
- Test package system
- Test server system
- Transport notes
- *Variables*

7.16.2 Notes

specific questions to answer

What happens when you click on the “run test” button:

- what processes start on the host
 - `java - jar /home/jenkins/slave.jar`, executing a shell running the contents of the job.xml “hudson.tasks.Shell/command” block:
 - * this block is labeled: “Execute shell: Command” in the “Build” section of the job, in the configure page for the job in the Jenkins user interface.
- what interface is used between the executing test (ultimately a bash shell script) and the jenkins processes
 - stop is performed by using the Jenkins REST API – by accessing “<http://localhost:8090/...stop>”
 - see Fuego-Jenkins

Each Jenkins node is defined in Jenkins in: `/var/lib/jenkins/nodes/config.xml`

- The name of the node is used as the “Device” and “NODE_NAME” for a test.
 - These environment variables are passed to the test agent, which is always “`java -jar /home/jenkins/slave.jar`”
- Who calls `ovgen.py`? The core does, at the very start of `main.sh` via the call to `set_overlay_vars` (which is in `overlays.sh:w`)

Jenkins calls:

- `java -jar /fuego-core/engine/slave.jar`
 - with variables:
 - * Device
 - * Reboot
 - * Rebuild
 - * Target_PreCleanup

- * Target_PostCleanup
 - * TESTDIR
 - * TESTNAME
 - * TESTSPEC
 - * FUEGO_DEBUG
- the “Test Run” section of the Jenkins job for a test has a shell script fragment with the following shell commands:

```
#logging areas=pre_test,pre_check,build,makepkg,deploy,snapshot,run,
#   post_test,processing, parser,criteria,charting
#logging levels=debug,verbose,info,warning,error
#export FUEGO_LOGLEVELS="run:debug,parser:verbose"
export FUEGO_CALLER="jenkins"
ftc run-test -b $NODE_NAME -t Functional.hello_world -s default \
  --timeout 6m \
  --reboot false \
  --rebuild false \
  --precleanup true \
  --postcleanup true
```

Some Jenkins notes:

Jenkins stores its configuration in plain files under JENKINS_HOME You can edit the data in these files using the web interface, or from the command line using manual editing (and have the changes take affect at runtime by selecting “Reload configuration from disk”.

By default, Jenkins assumes you are doing a continuous integration action of “build the product, then test the product”. It has default support for Java projects.

Fuego seems to use distributed builds (configured in a master/slave fashion).

Jenkins home has (from 2007 docs):

- config.xml - has stuff for the main user interface
- *.xml
- fingerprints - directory for artifact fingerprints
- jobs

– <JOBNAME>

- * config.xml
- * workspace
- * latest
- * builds
 - <ID>
 - build.xml
 - log
 - changelog.xml

The docker container interfaces to the outside host filesystem via the following links:

- /fuego-ro -> <host-fuego-location>/fuego-ro
- /fuego-rw -> <host-fuego-location>/fuego-rw
- /fuego-core -> <host-fuego-core-location>

What are all the fields in the “configure node” dialog: Specifically:

- where is “Description” used? - don’t know
- what is “# of executors”? - don’t know for sure
- how is “Remote root directory” used?
 - this is a path inside the Fuego container. I’m not sure what Jenkins uses it for.
- what are Labels used for?
 - as tags for grouping builds
- Launch method: Fuego uses the Jenkins option “Launch slave via execution of command on the Master” The command is “java -jar /fuego-core/engine/slave.jar”
 - NOTE: slave.jar comes from jta-core git repository, under engine/slave.jar

The fuego-core directory structure is:

```
overlays - has the base classes for fuego functions
base - has core shell functions
testplans - has json files for parameter specifications (deprecated)
distribs - has shell functions related to the distro
scripts - has fuego scripts and programs
(things like overlays.sh, loggen.py, parser/common.py, ovgen.py, etc.
slave.jar - java program that Jenkins calls to execute a test
tests - has a directory for each test
  Benchmark.foo
  Benchmark.foo.spec
  foo.sh
  test.yaml
  reference.log
  parser.py
  Functional.bar
  LTP
  etc.
```

What is groovy:

- an interpreted language for Java, used by the scriptler plugin to extend Jenkins

What plugins are installed with Jenkins in the JTA configuration?

- Jenkins Mailer, LDPA, External Monitor Job Type, PAM, Ant, Javadoc
- Jenkins Environment File (special)
- Credentials, SSH Credentials, Jenkins SSH Slags, SSH Agent
- Git Client, Subversion, Token Macro, Maven Integration, CVS
- Parameterized Trigger (special)
- Git, Groovy Label Assignment Extended Choie Parameter
- Rebuilder...

- Groovy Postbuild, ez-templates, HTML Publisher (special)
- JTA Benchmark show plot plugin (special)
- Log Parser Plugin (special)
- Dashboard view (special)
- Compact Columns (special)
- Jenkins Dynamic Parameter (special)
- flot (special) - benchmark graphs plotting plug-in for Fuego

Which of these did Cogent write?

- the flot plugin (not flot itself)

What scriptler scripts are included in JTA?

- getTargets
- getTestplans
- getTests

What language are scriptler scripts in?

- Groovy

What is the Maven plugin for Jenkins?

- Maven is an apache project to build and manage Java projects
 - I don't think the plugin is needed for Fuego

Jenkins refers to a "slave" - what does this mean?

- it refers to a sub-process that can be delegated work. Roughly speaking, Fuego uses the term 'target' instead of 'slave', and modifies the Jenkins interface to support this.

How the tests work

A simple test that requires no building is Functional.bc

- the test script and test program source are found in the directory: `/home/jenkins/tests/Functional.bc`

This runs a shell script on target to test the 'bc' program.

Functional.bc has the files:

```
bc-script.sh
declares "tarball=bc-script.tar.gz"
defines shell functions:
  test_build - calls 'echo' (does nothing)
  test_deploy - calls 'put bc-device.sh'
  test_run - calls 'assert_define', 'report'
              report references bc-device.sh
  test_processing - calls 'log_compare'
                  looking for "OK"
sources $JTA_SCRIPTS_PATH/functional.sh
bc-script.tar.gz
bc-script/bc-device.sh
```

Variables used (in bc-script.sh):

```
BOARD_TESTDIR
TESTDIR
FUNCTIONAL_BC_EXPR
FUNCTIONAL_BC_RESULT
```

A simple test that requires simple building:

Functional.synctest

This test tries to call fsync to write data to a file, but is interrupted with a kill command during the fsync(). If the child dies before the fsync() completes, it is considered success.

It requires shared memory (shmget, shmat) and semaphore IPC (semget and semctl) support in the kernel.

Functional synctest has the files:

```
synctest.sh
  declares "tarball=synctest.tar.gz"
  defines shell functions:
    test_build - calls 'make'
    test_deploy - calls 'put'
    test_run - calls 'assert_define', hd_test_mount_prepare, and 'report'
    test_processing - calls 'log_compare'
    looking for "PASS : sync interrupted"
  sources $JTA_SCRIPTS_PATH/functional.sh
synctest.tar.gz
  synctest/synctest.c
  synctest/Makefile
synctest_p.log
  has "PASS : sync interrupted"
```

Variables used (by synctest.sh)

```
CFLAGS
LDFLAGS
CC
LD
BOARD_TESTDIR
TESTDIR
FUNCTIONAL_SYNCTEST_MOUNT_BLOCKDEV
FUNCTIONAL_SYNCTEST_MOUNT_POINT
FUNCTIONAL_SYNCTEST_LEN
FUNCTIONAL_SYNCTEST_LOOP
```

Note: could be improved by checking for CONFIG_SYSVIPC in /proc/config.gz to verify that the required kernel features are present

MOUNT_BLOCKDEV and MOUNT_POINT are used by 'hd_test_mount_prepare' but are prefaced with FUNCTIONAL_SYNCTEST or BENCHMARK_BONNIE

from clicking "Run Test", to executing code on the target... config.xml has the slave command: /home/jenkins/slave.jar -> which is a link to /home/jenkins/jta/engine/slave.jar

overlays.sh has "run_python \$OF_OVGEN ..." where OF_OVGEN is set to "\$JTA_SCRIPTS_PATH/ovgen/ovgen.py"

How is overlays.sh called?

it is sourced by `/home/jenkins/scripts/benchmarks.sh` and
`/home/jenkins/scripts/functional.sh`

`functional.sh` is sourced by each `Functional.foo` script.

For `Functional.synctest`:

```
Functional.synctest/config.xml
  for the attribute <hudson.tasks.Shell> (in <builders>)
    <command>....
      souce $JTA_TESTS_PATH/$JOB_NAME/synctest.sh</command>

synctest.sh
'.' $JTA_SCRIPTS_PATH/functional.sh'
'source $JTA_SCRIPTS_PATH/overlays.sh'
'set_overlay_vars'
  (in overlays.sh)
  run_python $OF_OVGEN ($JTA_SCRIPTS_PATH/ovgen/ovgen.py) ...
    $OF_OUTPUT_FILE ($JTA_SCRIPTS_PATH/work/${NODE_NAME}_prolog.sh)
    generate xxx_prolog.sh
    SOURCE xxx_prolog.sh

functions.sh pre_test()

functions.sh build()
... test_build()

functions.sh deploy()

test_run()
  assert_define()
  functions.sh report()
```

NOTES about ovgen.py

What does this program do?

Here is a sample command line from a test console output:

```
python /home/jenkins/scripts/ovgen/ovgen.py \
  --classdir /home/jenkins/overlays//base \
  --ovfiles /home/jenkins/overlays//distrib/nologger.dist /home/jenkins/overlays//
↳boards/bbb.board \
  --testplan /home/jenkins/overlays//testplans/testplan_default.json \
  --specdir /home/jenkins/overlays//test_specs/ \
  --output /home/jenkins/work/bbb_prolog.sh
```

So, `ovgen.py` takes a `classdir`, a list of `ovfiles` a `testplan` and a `specdir`, and produces a `xxx_prolog.sh` file, which is then sourced by the main test script

Here is information about `ovgen.py` source:


```
Classes:
OFCClass
OFLayer
TestSpecs
```

```
Functions:
parseOFVars - parse Overlay Framework variables and definitions
parseVars - parse variables definitions
parseFunctionBodyName
parseFunction
baseParseFunction
parseBaseFile
parseBaseDir
parseInherit
parseInclude
parseLayerVarOverride
parseLayerFuncOverride
parseLayerVarDefinition
parseLayerCapList - look for BOARD.CAP_LIST
parseOverrideFile
generateProlog
generateSpec
parseGenTestPlan
parseSpec
parseSpecDir
run
```

Sample generated test script

bbb_prolog.sh is 195 lines, and has the following vars and functions:

```
from class:base-distrib:
    ov_get_firmware()
    ov_rootfs_kill()
    ov_rootfs_drop_caches()
    ov_rootfs_oom()
    ov_rootfs_sync()
    ov_rootfs_reboot()
    ov_rootfs_state()
    ov_logger()
    ov_rootfs_logread()

from class:base-board:
    LTP_OPEN_POSIX_SUBTEST_COUNT_POS
    MMC_DEV
    SRV_IP
    SATA_DEV
    ...
    JTA_HOME
    IPADDR
    PLATFORM=""
```

(continues on next page)

(continued from previous page)

```
LOGIN
PASSWORD
TRANSPORT
ov_transport_cmd()
ov_transport_put()
ov_transport_get()

from class:base-params:
    DEVICE
    PATH
    SSH
    SCP

from class:base-funcs:
    default_target_route_setup()

from testplan:default:
    BENCHMARK_DHRYSTONE_LOOPS
    BENCHMARK_<TESTNAME>_<VARNAME>
    ...
    FUNCTIONAL_<TESTNAME>_<VARNAME>
```

7.16.3 Logs

When a test is executed, several different kinds of logs are generated: devlog, systemlogs, the testlogs, and the console log.

created by Jenkins

- console log
 - this is located in `/var/lib/jenkins/jobs/<test_name>/builds/<build_id>/log`
 - is has the output from running the test script (on the host)

created by ftc

- console log
 - if ‘ftc’ was used to run the test, then the console log is created in the log directory, which is: `/fuego-rw/logs/<test_name>/<board>.<spec>.<build_id>.<build_number>/`
 - it is called `consolelog.txt`

created by the test script

- these are created in the directory: `/fuego-rw/logs/<test_name>/<board>.<spec>.<build_id>.<build_number>/`
- devlog has a list of commands run on the board during the test
 - named `devlog.txt`
- system logs have the log data from the board (e.g. `/var/log/messages`) before and after the test run:
 - named: `syslog.before.txt` and `syslog.after.txt`
- the test logs have the actual output from the test program on the target
 - this is completely dependent on what the test program outputs
 - named: `testlog.txt`
 - * this is the ‘raw’ log
 - there may be ‘parsed’ logs, which is the log filtered by `log_compare` operations:
 - * this is named: `testlog.p.txt` or `testlog.n.txt`
 - * the ‘p’ indicated positive results and the ‘n’ indicates negative results

7.16.4 Core scripts

The test script is sourced by the Fuego `main.sh` script

This script sources several other scripts, and ends up including `fuego_test.sh`

- load overlays and `set_overlay` vars
- `pre_test $TEST_DIR`
- build
- deploy
- `test_run`
- `set_testres_file`, `bench_processing`, `check_create_logrun` (if a benchmark)
- `get_testlog $TESTDIR`, `test_processing` (if a functional test)
- `get_testlog $TESTDIR` (if a stress test)
- `test_processing` (if a regular test)

functions available to test scripts: See Test Script APIs

Benchmark tests must provide a `parser.py` file, which extracts the benchmark results from the log data.

It does this by doing the following:

```
import common as plib
f = open(plib.TEST_LOG)
lines = f.readlines()
((parse the data))
```

This creates a dictionary with a key and value, where the key matches the string in the `reference.log` file

The `parser.py` program builds a dictionary of values by parsing the log from the test (basically the test output). It then sends the dictionary, and the pattern for matching the reference log test criteria to the routine: `common.py:process_data()`

It defines `ref_section_pat`, and passes that to `process_data()` Here are the different patterns for `ref_section_pat`:

```
9  "\\[[\\w]+.[gle]{2}\\]"
1  "\\[[\\w*].[gle]{2}\\]"
1  "^\\[[\\d\\w_ .]+.[gle]{2}\\]"
1  "^\\[[\\d\\w_ .-]+.[gle]{2}\\]"
1  "^\\[[\\w\\d&._/()]+.[gle]{2}\\]"
4  "^\\[[\\w\\d._]+.[gle]{2}\\]"
2  "^\\[[\\w\\d\\s_ .-]+.[gle]{2}\\]"
3  "^\\[[\\w\\d_ ./]+.[gle]{2}\\]"
5  "^\\[[\\w\\d_ .]+.[gle]{2}\\]"
1  "^\\[[\\w\\d_\\- .]+.[gle]{2}\\]"
1  "^\\[[\\w]+.[gle]{2}\\]"
1  "^\\[[\\w_ .]+.[gle]{2}\\]"
```

Why are so many different ones needed?? Why couldn't the syntax be: `<var-name> <test> <value>` on one line?

It turns out this is processed by an 'awk' script. thus the weird syntax. We should get rid of the awk script and use python instead.

How is benchmarking graphing done?

See Benchmark parser note

docker tips

See *docker tips*

7.17 License And Contribution Policy

7.17.1 License

Fuego has the following license policy.

Fuego consists of several parts, and includes source code from a number of different external test projects.

Default license

The default license for Fuego is the BSD 3-Clause license, as indicated in the LICENSE file at the top of the ‘fuego’ and ‘fuego-core’ source repositories.

If a file does not have an explicit license, or license indicator (such as SPDX identifier) in the file, then that file is covered by the default license for the project, with the exceptions noted below for “external test materials”.

When making contributions, if you do NOT indicate an alternative license for your contribution, the contribution will be assigned the license of the file to which the contribution applies (which may be the default license, if the file contains no existing license indicator).

Although we may allow for other licenses within the Fuego project in order to accommodate external software added to our system, our preference is to avoid the proliferation of licenses in the Fuego code itself.

External test materials

Individual tests in Fuego consist of files in the directory: `engine/tests/<test_name>` (which is known as the test home directory), and may include two types of materials:

1. Fuego-specific files
2. files obtained from external sources, which have their own license.

The Fuego-specific materials consist of files such as: `fuego_test.sh`, `spec.json`, `criteria.json`, `test.yaml`, `chart_config.json`, and possibly others as created for use in the Fuego project. External test materials may consist of tar files, helper scripts and patches against the source in the tar files.

Unless otherwise indicated, the Fuego-specific materials are licensed under the Fuego default license, and the external test materials are licensed under their own individual project license - as indicated in the test source.

In some cases, there is no external source code, but only source that is originally written for Fuego and stored in the test home directory. This commonly includes tests based on a single shell script, that is written to be deployed to the Device Under Test by `fuego_test.sh`. Unless otherwise indicated, these files (source and scripts) are licensed under the Fuego default license.

If there is any ambiguity in the category of a particular file (external or Fuego-specific), please designate the intended license clearly in the file itself, when making a contribution.

Copyright statements

Copyrights for individual contributions should be added to individual files, when the contributions warrant copyright assignment. Some trivial fixes to existing code may not need to have copyright assignment, and thus not every change to a file needs to include a copyright notice for the contributor.

License tags

Our preference is to use SPDX license identifier, rather than a license notice, to indicate the license of any materials in Fuego. Such identifiers and notices are only desired if the materials are not contributed under the default Fuego license of “BSD-3-Clause”.

In a `test.yaml`, please indicate the license of the upstream test program. If there is no upstream test program (ie, the test is self-contained), please specify the license of the Fuego test definition itself.

Please see <https://spdx.org/licenses/> for a list of SPDX license tags

Contributor agreement

The Fuego project does not require a signed Contributor License Agreement for contribution to the project. Instead, we utilize the following Developer Certificate of Origin that was copied from the Linux kernel.

Each contribution to Fuego must be accompanied by a Signed-off-by line in the patch or commit description, which indicates agreement to the following:

By making a contribution to this project, I certify that:

- (a) The contribution was created **in** whole **or in** part by me **and** I have the right to submit it under the **open** source license indicated **in** the file; **or**
- (b) The contribution **is** based upon previous work that, to the best of my knowledge, **is** covered under an appropriate **open** source license **and** I have the right under that license to submit that work **with** modifications, whether created **in** whole **or in** part by me, under the same **open** source license (unless I am permitted to submit under a different license), **as** indicated **in** the file; **or**
- (c) The contribution was provided directly to me by some other person who certified (a), (b) **or** (c) **and** I have **not** modified it.
- (d) I understand **and** agree that this project **and** the contribution are public **and** that a record of the contribution (including **all** personal information I submit **with** it, including my sign-off) **is** maintained indefinitely **and** may be redistributed consistent **with** this project **or** the **open** source license(s) involved.

Note: Please note that an “official” DCO at the web site <https://developercertificate.org/> has additional text (an LF copyright, address, and statement of non-copyability). All of these extra items are either nonsense or problematical in some legal sense. The above is a quote of a portion of the document found in the Linux kernel guide for submitting patches. See <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/process/submitting-patches.rst> (copied in March, 2018).

Each commit must include a DCO which looks like this

```
Signed-off-by: Joe Smith <joe.smith@email.com>
```

The project requires that the name used is your real name. Neither anonymous contributors nor those utilizing pseudonyms will be accepted.

You may type this line on your own when writing your commit messages. However, Git makes it easy to add this line to your commit messages. Make sure the user.name and user.email are set in your git configs. Use ‘-s’ or ‘-signoff’ options to ‘git commit’ to add the Signed-off-by line to the end of the commit message.

7.17.2 Submitting contributions

Please format contributions as a patch, and send the patch to the [Fuego mailing list](#)

Before making the patch, please verify that you have followed our preferred Coding style.

We follow the style of patches used by the Linux kernel, which is described here: <https://www.kernel.org/doc/html/latest/process/submitting-patches.html>

Not everything described there applies, but please do the following:

- Use a Signed-off-by line
- Send patch in plain text
- Include PATCH in the subject line
- Number patches in a series (1/n, 2/n, .. n/n)
- Use a subsystem prefix on the patch subject line
 - Patch subject should have: “subsystem: description”
 - In the case of modifications to a test, the subject should have: “test: description” (that is, the test is the subsystem name)
 - The test name can be the short name, if it is unambiguous
 - * That is, please don’t use the ‘Functional’ or ‘Benchmark’ prefix unless there are both types of tests with the same short name
- Describe your changes in the commit message body

Creating patches

If you use git, it’s easy to create a patch (or patch series), using `git format-patch`. Or, you can go directly to e-mailing a patch or patch series using `git send-email`.

Alternative submission method

I also allow patches as attachments to an e-mail to the list. This is something NOT supported by the Linux kernel community.

If the patch is too big (greater than 300K), then please add it to a public git repository, and let me know the URL for the repository. I can add a remote for the repo, and fetch it and cherry pick the patch. I prefer doing a fetch and cherry-pick to a pull request.

While I will sometimes process patches through a repo, it is strongly preferred for patches to go through the mailing list as plain text, so that community members can review the patch in public.

7.18 Core interfaces

This page documents the interface between the Jenkins front end and the Fuego core engine. See also [Variables](#)

7.18.1 From Jenkins to Fuego

Environment variables passed during a build

Built-in Jenkins variables for shell script jobs

BUILD_ID

The current test run id. As of (at least) Jenkins version 2.32.1, this is the same as the BUILD_NUMBER.

BUILD_NUMBER

The current test run number, such as “153”. This appears to be selected as the next numerical number in sequence, by the Jenkins system, at the start of a job.

BUILD_TAG

String of “jenkins-`${JOB_NAME}`-\${BUILD_NUMBER}”. Convenient to put into a resource file, a jar file, etc for easier identification.

BUILD_URL

Full URL of this test run, like <http://server:port/jenkins/job/foo/15/>

EXECUTOR_NUMBER

The unique number that identifies the current executor (among executors of the same machine) that’s carrying out this test run. This is the number you see in the “test executor status”, except that the number starts from 0, not 1.

JENKINS_HOME

The absolute path of the directory assigned on the master node for Jenkins to store data.

JENKINS_URL

Full URL of Jenkins, like <http://server:port/jenkins/>

JOB_NAME

Name of Jenkins job for this test. In Fuego, this will be something like: “myboard.default.Functional.foo” or “myboard.default.Benchmark.bar”. The job name has the form: `{board}.{spec}.{type}.{test_name}`

JOB_URL

Full URL of this test or test suite, like <http://server:port/jenkins/job/myboard.default.Functional.foo/>

NODE_LABELS

Whitespace-separated list of labels that are assigned to the node.

NODE_NAME

Name of the slave if the test run is on a slave, or “master” if run on master. In the case of Fuego, this is the board name (e.g. ‘beagleboneblack’)

WORKSPACE

The absolute path of the directory assigned to the test run as a workspace. For Fuego, this is always `/fuego-rw/buildzone`. Note that this comes from “custom workspace” setting in the job definition.

Fuego variables passed from Jenkins system configuration

The following variables are defined at the system level, and are passed by Jenkins in the environment of every job that is executed:

FUEGO_RO

The location where Fuego read-only data (configuration, boards and toolchains) are located. Currently /fuego-ro in the docker container.

FUEGO_RW

The location where Fuego read-write data is located (directories for builds, logs (run data), and other workspace and scratchpad areas). Currently /fuego-rw.

FUEGO_CORE

The location where the Fuego script system and tests are located. Currently /fuego-core.

Fuego variables passed from Jenkins job definition

These variables are defined in the job definition for a test:

Device

This is the target board to run the test on.

Reboot

Indicates to reboot the target device before running the test

Rebuild

Indicates that build instances of the test suite should be deleted, and the test suite rebuilt from the tarball

Target_PreCleanup

Indicates to clean up test materials left from a previous run of the test, before the test begins.

Target_PostCleanup

Indicates to clean up test materials after the test ends.

TESTPLAN

This has the name of the testplan used for this test job. Note that this selected by the end user from the output of getTestplans.groovy. (example value: testplan_default)

TESTNAME

Has the base name of the test (e.g. LTP vs Functional.LTP)

TESTSPEC

This has the name of the spec used for this test job

FUEGO_DEBUG

Can have a 1 to indicate verbose shell script output

7.18.2 From Fuego to Fuego

DISTRIB

Indicates the distribution file for the board. This comes from the board file. It's primary purpose is to select the logging features of the distribution on the target (to indicate whether there's a logger present on target). The value is often 'distrib/nologread.dist'

OF_BOARD_FILE

Full path to the board file

OF_CLASSDIR

full path to the overlay class directory (usually /home/jenkins/overlays//base)

OF_CLASSDIR_ARGS

argument specifying the overlay class directory (usually '-classdir /home/jenkins/overlays//base')

OF_DEFAULT_SPECDIR

path to directory containing test specs (usually /home/jenkins/overlays//test_specs)

OF_DISTRIB_FILE

path to the distribution file for the target board (often /home/jenkins/overlays//distrib/nologger.dist)

OF_OVFILES

FIXTHIS - document what OF_OVFILES is for

OF_OVFILES_ARGS

FIXTHIS - document what OF_OVFILES_ARGS is for

OF_ROOT

root directory for overlay generator (usually /home/jenkins/overlays/)

OF_SPECDIR_ARGS

argument to specify the test spec directory (usually '-specdir /home/jenkins/overlays//test_specs/')

OF_TESTPLAN_ARGS

OF_TESTPLAN

full path to the JSON test plan file for this test (often /home/jenkins/overlays//testplans/testplan_default.json)

OF_TESTPLAN_ARGS

argument specifying the path to the testplan (often '-testplan /home/jenkins/overlays//testplans/testplan_default.json')

TEST_HOME

home directory for the test materials for this test (example: /home/jenkins/tests/Functional.bc)

TESTDIR

base directory name of the test (example: Functional.bc)

TRIPLET

FIXTHIS - document TRIPLET

Deprecated

The following variables are no longer used in Fuego:

FUEGO_ENGINE_PATH

(deprecated in Fuego 1.1 - use '\$FUEGO_CORE/engine' now)

FUEGO_PARSER_PATH

(deprecated in Fuego 1.1)

7.18.3 Example Values

Here are the values from a run using the Jenkins front-end with job bbb.default.Functional.hello_world:

(these are sorted alphabetically):

```
AR=arm-linux-gnueabi-hf-ar
ARCH=arm
AS=arm-linux-gnueabi-hf-as
BUILD_DISPLAY_NAME=#2
BUILD_ID=2
BUILD_NUMBER=2
BUILD_TAG=jenkins-bbb.default.Functional.hello_world-2
BUILD_TIMESTAMP=2017-04-10_21-55-26
CC=arm-linux-gnueabi-hf-gcc
CONFIGURE_FLAGS=--target=arm-linux-gnueabi-hf --host=arm-linux-gnueabi-hf --build=x86_64-
↳ unknown-linux-gnu
CPP=arm-linux-gnueabi-hf-gcc -E
CROSS_COMPILE=arm-linux-gnueabi-hf-
CXX=arm-linux-gnueabi-hf-g++
CXXCPP=arm-linux-gnueabi-hf-g++ -E
EXECUTOR_NUMBER=0
FUEGO_CORE=/fuego-core
FUEGO_RO=/fuego-ro
FUEGO_RW=/fuego-rw
FUEGO_START_TIME=1491861326786
HOME=/var/lib/jenkins
HOST=arm-linux-gnueabi-hf
HUDSON_COOKIE=1b9620a3-d550-4cb1-afb1-9c5a29650c14
HUDSON_HOME=/var/lib/jenkins
HUDSON_SERVER_COOKIE=2334aa4d37eae7a4
JENKINS_HOME=/var/lib/jenkins
JENKINS_SERVER_COOKIE=2334aa4d37eae7a4
JOB_BASE_NAME=bbb.default.Functional.hello_world
JOB_DISPLAY_URL=http://unconfigured-jenkins-location/job/bbb.default.Functional.hello_
↳ world/display/redirect
JOB_NAME=bbb.default.Functional.hello_world
LD=arm-linux-gnueabi-hf-ld
LDFLAGS=--sysroot / -lm
LOGDIR=/fuego-rw/logs/Functional.hello_world/bbb.default.2.2
LOGNAME=jenkins
MAIL=/var/mail/jenkins
NODE_LABELS=bbb
NODE_NAME=bbb
```

(continues on next page)

(continued from previous page)

```
PATH=/usr/local/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
PREFIX=arm-linux-gnueabi
PWD=/fuego-rw/buildzone
RANLIB=arm-linux-gnueabi-ranlib
Reboot=false
Rebuild=true
RUN_CHANGES_DISPLAY_URL=http://unconfigured-jenkins-location/job/bbb.default.Functional.
↳hello_world/2/display/redirect?page=changes
RUN_DISPLAY_URL=http://unconfigured-jenkins-location/job/bbb.default.Functional.hello_
↳world/2/display/redirect
SDKROOT=/
SHELL=/bin/bash
SHLVL=3
Target_PostCleanup=true
Target_PreCleanup=true
TERM=xterm
TESTDIR=Functional.hello_world
TESTNAME=hello_world
TESTSPEC=default
USER=jenkins
WORKSPACE=/fuego-rw/buildzone
```

7.18.4 From Fuego to Jenkins

This section describes some of the operations that Fuego core scripts (or a test) can perform to invoke an action by Jenkins during a test. To perform a Jenkins action, Fuego uses Jenkins' REST API using the `wget` command.

- To abort a job, fuego does:
 - `wget -qO- ${BUILD_URL}/stop`
 - This is called by `common.sh: abort_job()`
- To check if another test instance is running (do a lock check), fuego does:
 - `wget -qO- "$(cat ${LOCKFILE})/api/xml?xpath=*/building/text%28%29"`
 - * `LOCKFILE` was previously set to hold the contents: `${BUILD_URL}`, so this resolves to:
 - `wget -qO- ${BUILD_URL}/api/xml?xpath=*/building/text()`
 - This is called by `functions.sh:concurrent_check()`

Jenkins python module

Fuego's `ftc` command uses the 'jenkins' python module to perform a number of operations with Jenkins. This module is used to:

- list nodes
- add nodes
- remove nodes
- list jobs
- build jobs

- remove jobs
- re-synch build data for a job (using `get_job_config()` and `reconfig_job()`)
- add view

Note: Fuego uses jenkins-cli to add jobs (described next).

Jenkins-cli interface

You can run Jenkins commands from the command line, using the pre-installed jenkins-cli interface. This is used by Fuego's `ftc` command to create jobs.

jenkins-cli.jar is located in the Docker container at:

```
/var/cache/jenkins/war/WEB-INF/jenkins-cli.jar
```

See <https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI> for information about using this plugin.

Here is a list of available commands for this plugin:

```
build
  Runs a test, and optionally waits until its completion.
cancel-quiet-down
  Cancel the effect of the "quiet-down" command.
clear-queue
  Clears the test run queue
connect-node
  Reconnect to a node
console
  Retrieves console output of a build
copy-job
  Copies a test.
create-job
  Creates a new test by reading stdin as a configuration XML file.
delete-builds
  Deletes test record(s).
delete-job
  Deletes a test
delete-node
  Deletes a node
disable-job
  Disables a test
disconnect-node
  Disconnects from a node
enable-job
  Enables a test
get-job
  Dumps the test definition XML to stdout
groovy
  Executes the specified Groovy script.
groovysh
  Runs an interactive groovy shell.
```

(continues on next page)

(continued from previous page)

```
help
  Lists all the available commands.
install-plugin
  Installs a plugin either from a file, an URL, or from update center.
install-tool
  Performs automatic tool installation, and print its location to
  stdout. Can be only called from inside a test run.
keep-build
  Mark the test run to keep the test run forever.
list-changes
  Dumps the changelog for the specified test(s).
list-jobs
  Lists all tests in a specific view or item group.
list-plugins
  Outputs a list of installed plugins.
login
  Saves the current credential to allow future commands to run
  without explicit credential information.
logout
  Deletes the credential stored with the login command.
mail
  Reads stdin and sends that out as an e-mail.
offline-node
  Stop using a node for performing test runs temporarily, until the
  next "online-node" command.
online-node
  Resume using a node for performing test runs, to cancel out the
  earlier "offline-node" command.
quiet-down
  Quiet down Jenkins, in preparation for a restart. Don't start
  any test runs.
reload-configuration
  Discard all the loaded data in memory and reload everything from
  file system. Useful when you modified config files directly on disk.
restart
  Restart Jenkins
safe-restart
  Safely restart Jenkins
safe-shutdown
  Puts Jenkins into the quiet mode, wait for existing test runs to
  be completed, and then shut down Jenkins.
session-id
  Outputs the session ID, which changes every time Jenkins restarts
set-build-description
  Sets the description of a test run.
set-build-display-name
  Sets the displayName of a test run
set-build-result
  Sets the result of the current test run. Works only if invoked
  from within a test run.
shutdown
  Immediately shuts down Jenkins server
```

(continues on next page)

(continued from previous page)

```
update-job
    Updates the test definition XML from stdin.
    The opposite of the get-job command
version
    Outputs the current version.
wait-node-offline
    Wait for a node to become offline
wait-node-online
    Wait for a node to become online
who-am-i
    Reports your credential and permissions
```

Scripts to process Fuego data

Benchmark parsing

In Fuego, Benchmark log parsing is done by a python system consisting of `parser.py` (from each test), `dataload.py` and utility functions in `fuego-core/engine/scripts/parser`

See Benchmark parser notes, [parser.py](#), `reference.log` and [Parser module API](#).

Postbuild action

In Fuego, Jenkins jobs are configured to perform a postbuild action, to set the description of a test with links to the test log (and possibly plot and other files generated in post-processing)

7.19 Adding a new test

7.19.1 Overview of Steps

To add a new test to Fuego, you need to perform the following steps:

- 1. Decide on a test name and type
- 2. Make the test directory
- 3. Get the source (or binary) for the test
- 4. Write a test script for the test
- 5. Add the `test_specs` (if any) for the test
- 6. Add log processing to the test
- 6-a. (if a benchmark) Add `parser.py` and criteria and reference files
- 7. Create the Jenkins test configuration for the test

7.19.2 Decide on a test name

The first step to creating a test is deciding the test name. There are two types of tests supported by Fuego: functional tests and benchmark tests. A functional test either passes or fails, while a benchmark test produces one or more numbers representing some performance measurements for the system.

Usually, the name of the test will be a combination of the test type and a name to identify the test itself. Here are some examples: *bonnie* is a popular disk performance test. The name of this test in the fuego system is *Benchmark.bonnie*. A test which runs portions of the posix test suite is a functional test (it either passes or fails), and in Fuego is named *Functional.posixtestsuite*. The test name should be all one word (no spaces).

This name is used as the directory name where the test materials will live in the Fuego system.

7.19.3 Create the directory for the test

The main test directory is located in `/fuego-core/engine/tests/<test_name>`

So if you just created a new Functional test called ‘foo’, you would create the directory:

- `/fuego-core/engine/tests/Functional.foo`

7.19.4 Get the source for a test

The actual creation of the test program itself is outside the scope of Fuego. Fuego is intended to execute an existing test program, for which source code or a script already exists.

This page describes how to integrate such a test program into the Fuego test system.

A test program in Fuego is provided in source form so that it can be compiled for whatever processor architecture is used by the target under test. This source may be in the form of a tarfile, or a reference to a git repository, and one or more patches.

Create a tarfile for the test, by downloading the test source manually, and creating the tarfile. Or, note the reference for the git repository for the test source.

tarball source

If you are using source in the form of a tarfile, you add the name of the tarfile (called ‘tarball’) to the test script.

The tarfile may be compressed. Supported compression schemes, and their associated extensions are:

- uncompressed (extension=’.tar’)
- compressed with gzip (extension=’.tar.gz’ or ‘.tgz’)
- compressed with bzip2 (extension=’.bz2’)

For example, if the source for your test was in the tarfile ‘foo-1.2.tgz’ you would add the following line to your test script, to reference this source:

```
tarball=foo-1.2.tgz
```


git source

If you are using source from an online git repository, you reference this source by adding the variables ‘gitrepo’ and ‘gitref’ to the test script.

In this case, the ‘gitrepo’ is the URL used to access the source, and the ‘gitref’ refers to a commit id (hash, tag, version, etc.) that refers to a particular version of the code.

For example, if your test program is built from source in an online ‘foo’ repository, and you want to use version 1.2 of that (which is tagged in the repository as ‘v1.2’, on the master branch, you might have some lines like the following in the test’s script.

```
gitrepo=http://github.com/sampleuser/foo.git
gitref=master/v1.2
```

script-based source

Some tests are simple enough to be implemented as a single script (that runs on the board). For these tests, no additional source is necessary, and the script can just be placed directly in the test’s home directory. In *fuego_test.sh* you must set the following variable:

```
local_source=1
```

During the deploy phase, the script is sent to the board directly from the test home directory instead of from the test build directory.

7.19.5 Test script

The test script is a small shell script called *fuego_test.sh*. It specifies the source tarfile containing the test program, and provides implementations for the functions needed to build, deploy, execute, and evaluate the results from the test program.

The test script for a functional test should contain the following:

- source reference (either tarball or gitrepo and gitref)
- function test_pre_check (optional)
- function test_build
- function test_deploy
- function test_run
- function test_processing

The test_pre_check function is optional, and is used to check that the test environment and target configuration and setup are correct in order to run the test.

Sample test script

Here is the `fuego_test.sh` script for the test `Functional.hello_world`. This script demonstrates a lot of the core elements of a test script.:

```
#!/bin/bash

tarball=hello-test-1.0.tgz

function test_build {
    make
}

function test_deploy {
    put hello $BOARD_TESTDIR/fuego.$TESTDIR/
}

function test_run {
    report "cd $BOARD_TESTDIR/fuego.$TESTDIR; \
    ./hello $FUNCTIONAL_HELLO_WORLD_ARG"
}

function test_processing {
    log_compare "$TESTDIR" "1" "SUCCESS" "p"
}
```

Description of base test functions

The base test functions (`test_build`, `test_deploy`, `test_run`, and `test_processing`) are fairly simple. Each one contains a few statements to accomplish that phase of the test execution.

You can find more information about each of these functions at the following links:

- `test_pre_check`
- `test_build`
- `test_deploy`
- `test_run`
- `test_processing`

7.19.6 Test spec and plan

Another element of every test is the *test spec*. A file is used to define a set of parameters that are used to customize the test for a particular use case.

You must define the test spec(s) for this test, and add an entry to the appropriate testplan for it.

Each test in the system must have a test spec file. This file is used to list customizable variables for the test.

If a test program has no customizable variables, or none are desired, then at a minimum a *default* test spec must be defined, with no test variables.

The test spec file is:

- named ‘spec.json’ in the test directory,

- in JSON format,
- provides a `testName` attribute, and a `specs` attribute, which is a list,
- may include any named spec you want, but must define at least the ‘default’ spec for the test
 - Note that the ‘default’ spec can be empty, if desired.

Here is an example one that defines no variables.

```
{
  "testName": "Benchmark.OpenSSL",
  "specs": {
    "default": {}
  }
}
```

And here is the spec.json of the Functional.hello_world example, which defines three specs:

```
{
  "testName": "Functional.hello_world",
  "specs": {
    "hello-fail": {
      "ARG": "-f"
    },
    "hello-random": {
      "ARG": "-r"
    },
    "default": {
      "ARG": ""
    }
  }
}
```

Next, you may want to add an entry to one of the testplan files. These files are located in the directory `/fuego-core/engine/overlays/testplans`.

Choose a testplan you would like to include this test, and edit the corresponding file. For example, to add your test to the list of tests executed when the ‘default’ testplan is used, add an entry `default` to the ‘testplan_default.json’ file.

Note that you should add a comma after your entry, if it is not the last one in the list of *tests*.

Please read Test Specs and Plans for more details.

7.19.7 Test results parser

Each test should also provide some mechanism to parse the results from the test program, and determine the success of the test.

For a simple Functional test, you can use the `log_compare` function to specify a pattern to search for in the test log, and the number of times that pattern should be found in order to indicate success of the test. This is done from the `test_processing` function in the test script.

Here is an example of a call to `log_compare`:

```
function test_processing {
    log_compare "$TESTDIR" "11" "^TEST.*OK" "p"
}
```

This example looks for the pattern `^TEST.*OK`, which finds lines in the test log that start with the word ‘TEST’ and are followed by the string ‘OK’ on the same line. It looks for this pattern 11 times.

`log_compare` can be used to parse the logs of simple tests with line-oriented output.

For tests with more complex output, and for Benchmark tests that produce numerical results, you must add a python program called ‘`parser.py`’, which scans the test log and produces a data structure used by other parts of the Fuego system.

See `parser.py` for information about this program.

7.19.8 Pass criteria and reference info

You should also provide information to Fuego to indicate how to evaluate the ultimate resolution of the test.

For a Functional test, it is usually the case that the whole test passes only if all individual test cases in the test pass. That is, one error in a test case indicates overall test failure. However, for Benchmark tests, the evaluation of the results is more complicated. It is required to specify what numbers constitute success vs. failure for the test.

Also, for very complicated Functional tests, there may be complicated results, where, for example, some results should be ignored.

You can specify the criteria used to evaluate the test results, by creating a ‘*criteria.json*’ file for the test.

Finally, you may wish to add a file that indicates certain information about the test results. This information is placed in the ‘*reference.json*’ file for a test.

Please see the links for those files to learn more about what they are and how to write them, and customize them for your system.

7.19.9 Jenkins job definition file

The last step in creating the test is to create the Jenkins job for it.

A Jenkins job describes to Jenkins what board to run the test on, what variables to pass to the test (including the test spec (or variant), and what script to run for the test.

Jenkins jobs are created using the command-line tool ‘`ftc`’.

A Jenkins job has the name `<node_name>.<spec>.<test_type>.<test_name>`

You create a Jenkins job using a command like the following:

- `$ ftc add-jobs -b myboard -t Functional.mytest [-s default]`

The `ftc` ‘`add-jobs`’ sub-command uses ‘`-b`’ to specify the board, ‘`-t`’ to specify the test, and ‘`-s`’ to specify the test spec that will be used for this Jenkins job.

In this case, the name of the Jenkins job that would be created would be:

- `myboard.default.Functional.mytest`

This results in the creation of a file called `config.xml`, in the `/var/lib/jenkins/jobs/<job_name>` directory.

7.19.10 Publishing the test

Tests that are of general interest should be submitted for inclusion into fuego-core.

Right now, the method of doing this is to create a commit and send that commit to the fuego mailing list, for review, and hopefully acceptance and integration by the fuego maintainers.

In the future, a server will be provided where test developers can share tests that they have created in a kind of “test marketplace”. Tests will be available for browsing and downloading, with results from other developers available to compare with your own results. There is already preliminary support for packaging a test using the ‘ftc package-test’ feature. More information about this service will be made available in the future.

7.19.11 Technical Details

This section has technical details about a test.

Directory structure

The directory structure used by Fuego is documented at [\[\[Fuego directories\]\]](#)

Files

A test consists of the following files or items:

File or item	format	location	description	test type
con-fig.xml	Jenkins XML	/var/lib/jenkins/jobs/{test_name}	Has the Jenkins (front-end) configuration for the test	all
tarfile	tar format	/fuego-core/engine/tests/{test_name}	Has the source code for the test program	all
patches	patch format	/fuego-core/engine/tests/{test_name}	Zero or more patches to customize the test program (applied during the unpack phase)	all
base script	shell script	/fuego-core/engine/tests/{test_name}/fuego_test.sh	Is the shell script that implements the different test phases in Fuego	all
test spec	JSON	/fuego-core/engine/tests/{test_name}/spec.json	Has groups of variables (and their values) that can be used with this test	all
test plan(s)	JSON	/fuego-core/engine/overlays/testplan	Has the testplans for the entire system	all
parser.py	python	/fuego-core/engine/tests/{test_name}	Python program to parse benchmark metrics out of the log, and provide a dictionary to the Fuego plotter all,	all but can be missing for functional tests
pass criteria	JSON	/fuego-core/engine/tests/{test_name}/criteria.json	Has the “pass” criteria for the test	all
reference info	JSON	/fuego-core/engine/tests/{test_name}/reference.json	Has additional information about test results, such as the units for benchmark measurements	benchmark only
reference.log	Fuego-specific	/fuego-core/engine/tests/{test_name}	Has the threshold values and comparison operators for benchmark metrics measured by the test	benchmark only
(deprecated)	p/n logs-text	/fuego-core/engine/tests/{test_name}	Are logs with the results (positive or negative) passed out, for determination of test pass/fail	functional only

7.20 Using Batch Tests

A “batch test” in Fuego is a Fuego test that runs a series of other tests as a group. The results of the individual tests are consolidated into a list of testcase results for the batch test.

Prior to Fuego version 1.5, there was a different feature, called “testplans”, which allowed users to compose sets of tests into logical groups, and run them together. The batch test system, introduced in Fuego version 1.5 replaces the testplan system.

7.20.1 How to make a batch test

A batch test consists of a Fuego test that runs other tests. A Fuego batch test is similar to other Fuego tests, in that the test definition lives in `fuego-core/tests/<test-name>`, and it consists of a `fuego_test.sh` file, a spec file, a `parser.py`, a `test.yaml` file and possibly other files.

The difference is that a Fuego batch test runs other Fuego tests, as a group. The batch test has a few elements that are different from other tests.

Inside the `fuego_test.sh` file, a batch test must define two main elements:

1. The testplan element
2. The `test_run` function, with commands to run other tests

Testplan element

The testplan element consists of data assigned to the shell variable `BATCH_TESTPLAN`. This variable contains lines that specify, in machine-readable form, the tests that are part of the batch job. The testplan is specified in json format, and is used to specify the attributes (such as timeout, flags, and specs) for each test. The testplan element is used by `ftc add-jobs` to create Jenkins jobs for each sub-test that is executed by this batch test.

The `BATCH_TESTPLAN` variable must be defined in the `fuego_test.sh` file. The definition must begin with a line starting with the string `'BATCH_TESTPLAN='` and end with a line starting with the string `'END_TESTPLAN'`. By convention this is defined as a shell “here document”, like this example:

```
BATCH_TESTPLAN=$(cat <<END_TESTPLAN
{
    "testPlanName": "foo_plan",
    "tests": [
        { "testName": "Functional.foo" },
        { "testName": "Functional.bar" }
    ]
}
END_TESTPLAN
)
```

The lines of the testplan follow the format described at [Testplan_Reference](#). Please see that page for details about the plan fields and structure (the schema for the testplan data).

test_run function

The other element in a batch test's `fuego_test.sh` is a `test_run` function. This function is used to actually execute the tests in the batch.

There are two functions that are available to help with this:

- `allocate_next_batch_id`
- `run_test`

The body of the `test_run` function for a batch test usually has a few common elements:

- setting of the `FUEGO_BATCH_ID`
- execution of the sub-tests, using a call to the function `run_test` for each one

Here are the commands in the `test_run` function for the test `Functional.batch_hello`:

```
function test_run {
    export TC_NUM=1
    DEFAULT_TIMEOUT=3m
    export FUEGO_BATCH_ID="hello-
$(allocate_next_batch_id)"

    # don't stop on test errors
    set +e
    log_this "echo \"batch_id=$FUEGO_BATCH_ID\""
    TC_NAME="default"
    run_test Functional.hello_world
    TC_NAME="fail"
    run_test Functional.hello_world -s hello-fail
    TC_NAME="random"
    run_test Functional.hello_world -s hello-random
    set -e
}
```

Setting the batch_id

Fuego uses a `batch_id` to indicate that a group of test runs are related. Since a single Fuego test can be run in many different ways (e.g. from the command line or from Jenkins, triggered manually or automatically, or as part of one batch test or another), it is helpful for the run data for a test to be assigned a `batch_id` that can be used to generate reports or visualize data for the group of tests that are part of the batch.

A batch test should set the `FUEGO_BATCH_ID` for the run to a unique string for that run of the batch test. Each sub-test will store the batch id in its `run.json` file, and this can be used to filter run data in subsequent test operations. The Fuego system can provide a unique number, via the routine `allocate_next_batch_id`. By convention, the `batch_id` for a test is created by combining a test-specific prefix string with the number returned from `allocate_next_batch_id`.

In the example above, the prefix used is 'hello-', and this would be followed by a number returned by the function `allocate_next_batch_id`.

Executing sub-tests

The `run_test` function is used to execute the sub-tests that are part of the batch. The other portions of the example above show setting various shell variables that are used by `run_test`, and turning off 'errexit' mode while the sub-tests are running.

In the example above, `TC_NUM`, `TC_NAME`, and `DEFAULT_TIMEOUT` are used for various effects. These variables are optional, and in most cases a batch test can be written without having to set them. Fuego will generate automatic strings or values for these variables if they are not defined by the batch test.

Please see the documentation for `run_test` for details about the environment and arguments used when calling the function.

Avoiding stopping on errors

The example above shows use of `set +e` and `set -e` to control the shell's 'errexit' mode. By default, Fuego runs tests with the shell errexit mode enabled. However, a batch test should anticipate that some of its sub-tests might fail. If you want all of the tests in the batch to run, even if some of them fail, they you should use `set +e` to disable errexit mode, and `set -e` to re-enable it when you are done.

Of course, if you want the batch test to stop if one of the sub-tests fails, they you should control the errexit mode accordingly (for example, leaving it set during all sub-test executions, or disabling it or enabling it only during the execution of particular sub-tests).

Whether to manipulate the shell errexit mode or not depends on what the batch test is doing. If it is implementing a sequence of dependent test stages, the errexit mode should be left enabled. If a batch test is implementing a series of unrelated, independent tests, the errexit mode should be disabled and re-enabled as shown.

7.20.2 Test output

The `run_test` function logs test results in a format similar to TAP13. This consists of the test output, followed by a line starting with the batch id (inside double brackets), then "ok" or "not ok" to indicate the sub-test result, followed by the testcase number and testcase name.

A standard `parser.py` for this syntax is available and used by other batch tests in the system (See `fuego-core/tests/Functional.batch_hello/parser.py`)

7.20.3 Preparing the system for a batch job

In order to run a batch test from Jenkins, you need to define a Jenkins job for the batch test, and jobs for all of the sub-tests that are called by the batch test.

You can use `ftc add-jobs` with the batch test, and Fuego will create the job for the batch test itself as well as jobs for all of its sub-tests.

It is possible to run a batch test from the command line using 'ftc run-test', without creating Jenkins jobs. However if you want to see the results of the test in the Jenkins interface, then the Jenkins test jobs need to be defined prior to running the batch test from the command line.

7.20.4 Executing a batch test

A batch test is executed the same way as any other Fuego test. Once installed as a Jenkins job, you can execute it using the Jenkins interface (manually), or use Jenkins features to cause it to trigger automatically. Or, you can run the test from the command line using `ftc run-test`.

7.20.5 Viewing batch test results

You can view results from a batch test in two ways:

1. Inside the Jenkins interface, or
2. Using `ftc` to generate a report.

Jenkins batch test results tables

Inside the Jenkins interface, a batch job will display the list of sub-tests, and the PASS/FAIL status of each one. In addition, if there is a Jenkins job associated with a particular sub-test, there will be a link in the table cell for that test run, that you can click to see that individual test's result and data in the Jenkins interface.

Generating a report

You can view a report for a batch test, by specifying the `batch_id` with the `ftc gen-report` command.

To determine the `batch_id`, look at the log for the batch test (`testlog.txt` file). Or, generate a report listing the `batch_ids` for the batch test, like so:

```
$ ftc gen-report --where test=batch_<name> --fields timestamp,batch_id
```

Select an appropriate `batch_id` from the list that appears, and note it for use in the next command.

Now, to see the results from the individual sub-tests in the batch, use the desired `batch_id` as part of a “where” clause, like so:

```
$ ftc gen-report --where batch_id=<batch_id>
```

You should see a report with all sub-test results for the batch.

7.20.6 Miscellaneous notes

Timeouts

The timeout for a batch test should be long enough for all sub-tests to complete. When a batch test is launched from Jenkins, the board on which it will run is reserved and will be unavailable for tests until the entire batch is complete. Keep this in mind when executing batch tests that call sub-tests that have a long duration.

The timeout for individual sub-tests can be specified multiple ways. First, the timeout listed in the testplan (embedded in `fuego_test.sh` as the `BATCH_TESTPLAN` variable) is the one assigned to the Jenkins job for the sub-test, when jobs are created during test installation into Jenkins. These take effect when a sub-test is run independently from the batch test.

If you want to specify a non-default timeout for a test, then you must use a `--timeout` argument to the `run_test` function, for that sub-test.

7.21 Parser module API

The file `common.py` is the python module for performing benchmark log file processing, and results processing and aggregation.

It is used by the `parser.py` program from the test directory, to process the log after each test run. The data from a test run is processed to:

- Check numeric values for pass/fail result(by checking against a reference threshold values)
- Determine the overall result of the test, based on potentially complex results criteria
- Save the data for use in history and comparison charts

7.21.1 Parser API

The following are functions used during log processing, by a test's parser.py program.

- `parse_log()` - parse the data from a test log
 - This routine takes a regular expression, with one or more groups, and results a list of tuples for lines that matched the expression
 - The tuples consist of the strings from the matching line corresponding to the regex groups
- `process()` - process results from a test
 - This routine takes a dictionary of test results, and does 3 things:
 - * Formats them into the run.json file (run results file)
 - * Detects pass or fail by using the specified pass criteria
 - * Formats the data into charts (plots and tables)
- `split_output_per_testcase()`
 - Split testlog into chunks accessible from the Jenkins user interface (one per testcase)

In general, a parser module will normally call `parse_log()`, then take the resulting list of matching groups to construct a dictionary to pass to the `process()` routine.

If the log file format is amendable, the parser module may also call `split_output_per_testcase()` to generate a set of files from the testlog, that can be referenced from the charts generated by the charting module.

Please see parser.py for more details and examples of use of the API.

7.21.2 Deprecated API

Note: The following information is for historical purposes only. Although the API is still present in Fuego, these APIs are deprecated.

In Fuego version 1.1 and prior, the following functions were used. These are still available for backwards compatibility with tests written for these versions of Fuego.

- `parse()`
- `process_data()`

(see parser.py for invocation details)

`parse()`

- input:
 - `cur_search_pattern` - compiled re search pattern
- output:
 - list of regular expression matches for each line matching the specified pattern

This routine scans the current log file, using a regular expression. It returns an re match object for each line of the log file that matches the expression.

This list is used to populate a dictionary of metric/value pairs that can be passed to the `process_data` function.

process_data

This is the main routine of the module. It processes the list of metrics, and populates various output files for test.

- input:
 - ref_section_pat - regular expression used to read reference.log
 - cur_dict - dictionary of metric/value pairs
 - m - indicates the size of the plot. It should be one of: 's', 'm', 'l', 'xl'
 - * if 'm', 'l', or 'xl' are used, then a multiplot is created
 - label - label for the plot

This routine has the following outline:

- write_report_results
- read the reference thresholds
- check the values against the reference thresholds
- store the plot data to a file (plot.data)
- create the plot
- save the plot to an image file (plot.png)

7.21.3 Developer notes

functions in common.py

- hls - print a big warning or error message
- parse_log(regex_str) - specify a regular expression string to use to parse lines in the log
 - this is a helper function that returns a list of matches (with groups) that the parser.py can use to populate its dictionary of measurements
- parse(regex_compiled_object)
 - similar to parse_log, but it takes a compiled regular expression object, and returns a list of matches (with groups)
 - this is deprecated, but left to support legacy tests
- split_tguid()
- split_test_id()
- get_test_case()
- add_results()
- init_run_data()
- get_criterion()
- check_measure()
- decide_status()
- convert_reference_log_to_criteria()
- load_criteria()

- `apply_criteria()`
- `create_default_ref()`
- `prepare_run_data()`
- `extract_test_case_ids()`
- `update_results_json()`
- `delete()`
- `save_run_json()`
- `process(results)`
 - `results` is a dictionary with
 - * `key=test_case_id` (not including measure name)
 - for a functional test, the `test_case_id` is usually “default.<test_name>”
 - * `value=list of measures` (for a benchmark)
 - * or `value=string (PASS|FAIL|SKIP)` (for a functional test)
- `process_data(ref_sections_pat, test_results, plot_type, label)`

call trees

```

process_data(ref_section_pat, test_results, plot_type, label)
  process_data(measurements)
    prepare_run_data(results)
      run_data = (prepare non-results data structure)
      ref = read reference.json
      or ref = create_default_ref(results)
      init_run_data(run_data, ref)
        (put ref into run_data structure)
        (mark some items as SKIP)
      add_results(results, run_data)
        for each item in results dictionary:
          (check for results type: list or str)
          if list, add measure
          if str, set status for test_case
      apply_criteria(run_data)
        load_criteria()
          (load criteria.json)
          or convert_reference_log_to_criteria()
        check_measure()
          get_criterion()
        decide_status()
          get_criterion()
      save_run_json(run_data)
      update_results_json()
      (return appropriate status)

```

miscellaneous notes

- `create_default_ref_tim` (for `docker.hello-fail.Functional.hello_world`)
 - `ref={ 'test_sets': [{ 'test_cases': [{ 'measurements': [{ 'status': 'FAIL', 'name': 'Functional' }], 'name': 'default' }], 'name': 'default' }}`
- `create_default_ref`
 - `ref={ 'test_sets': [{ 'test_cases': [{ 'status': 'FAIL', 'name': 'default' }], 'name': 'default' }}`

data format and tguid rules

The current API and the old parser API take different data and allow different test identifiers. This sections explains the difference:

Data format for benchmark test with new API

- `measurements[test_case_id] = [{"name": measure_name, "measure": value}]`

Data format for benchmark test with old API:

- in `reference.log`
 - if tguid is a single word, then use that word as the measure name and “default” as the test_case.
 - * e.g. for `benchmark.arm`, the `reference.log` has “short”. This becomes the fully-qualified tguid: `arm.default.arm.short`:
 - `test_name = arm, test_case = default, test_case_id = arm, measure = short`

Data format for functional tests with new API and the old API is the same:

- e.g. `measurements["status"] = "PASS|FAIL"`

7.22 FAQ

Here is a list of Frequently Asked Questions and Answers about Fuego:

7.22.1 Languages and formats used

Q. Why does Fuego use shell scripting as the language for tests?

There are other computer languages which have more advanced features (such as data structions, object orientation, rich libraries, concurrency, etc.) than shell scripting. It might seem odd that shell scripting was chosen as the language for implementing the base scripts for the tests in fuego, given the availability of these other languages.

The Fuego architecture is specifically geared toward host/target testing. In particular, tests often perform a variety of operations on the target in addition to the operations that are performed on the host. When the base script for a test runs on the host machine, portions of the test are invoked on the target. It is still true today that the most common execution environment (besides native code) that is available on almost every embedded Linux system is a POSIX-compliant shell. Even devices with very tight memory requirements usually have a busybox ‘ash’ shell available.

In order to keep the base script consistent, Fuego uses shell scripting on both the host and target systems. Shell operations are performed on the target using ‘cmd’, ‘report’ and ‘report_append’ functions provided by Fuego.

Note that Fuego officially use ‘bash’ as the shell on the host, but does not require a particular shell implementation to be available on the target. Therefore, it is important to use only POSIX-compatible shell features for those aspects of the test that run on target.

7.23 Fuego naming rules

To add boards or write tests for Fuego, you have to create a number of files and define a number of variables.

Here are some rules and conventions for naming things in Fuego:

7.23.1 Fuego test name

- a Fuego test name must have one of the following prefixes:
 - ‘Functional.’
 - ‘Benchmark.’
- the name following the prefix is known as the base test name, and has the following rules:
 - it may only use letters, numbers and underscores
 - * that is - no dashes
 - it may use upper and lower case letters
- All test definition materials reside in a directory with the full test name:
 - e.g. Functional.hello_world

7.23.2 Test files

- the base script file has the name **fuego_test.sh**
- the default spec file for a test has name **spec.json**
- the default criteria file for a test has the name **criteria.json**
- the reference file for a test has the name **reference.json**
- the parser module for a test has the filename **parser.py**

7.23.3 Test spec names

Test spec names are declared in the spec file for a test. These names should consist of letters, numbers and underscores only. That is, no periods should be used in spec names.

Every test should have at least one spec, called the ‘default’ spec. If no spec file exists for a test, then Fuego generates a ‘default’ spec (on-the-fly), which is empty (ie. has no test variables).

7.23.4 Board names

- boards are defined by files in the /fuego-ro/boards directory
- their filenames consists of the board name, with the suffix “.board”
 - e.g. beaglebone.board
- a board name should have only letters, numbers and underscores.
 - specifically, no dashes, periods, or other punctuation is allowed

7.23.5 Jenkins element names

Several of the items in Fuego are represented in the Jenkins interface. The following sections describe the names used for these elements.

Node name

- A Jenkins node corresponding to a board must have the same name as the board.
 - e.g. beaglebone

Job name

- A Jenkins job is used to execute a test.
- Jenkins job names should consist of these parts: <board>.<spec>.<test_name>
 - e.g. beaglebone.default.Functional.hello_world

7.23.6 Run identifier

A Fuego run identifier is used to refer to a “run” of a test - that is a particular invocation of a test and it’s resulting output, logs and artifacts. A run identifier should be unique throughout the world, as these are used in servers where data from runs from different hosts are stored.

The parts of a run id are separated by dashes, except that the separator between the host and the board is a colon.

A fully qualified (global) run identifier consist of the following parts:

- test name
- spec name
- build number
- the word *on*
- host
- board

FIXTHIS - global run ids should include timestamps to make them globally unique for all time

Example: Functional.LTP-quickhit-3-on-timdesk:beaglebone

A shortened run identifier may omit the *on* and *host*. This is referred to as a local run id, and is only valid on the host where the run was produced.

Example:

- Functional.netperf-default-2-minnow

7.23.7 timestamp

- A Fuego timestamp has the format: YYYY-mm-dd_HH:MM:SS
 - e.g. 2017-03-29_10:25:14
- times are expressed in localtime (relative to the host where they are created)

7.23.8 test identifiers

Also known as TGUIDs (or test globally unique identifiers), a test identifier refers to a single unit of test operation or result from the test system. A test identifier may refer to a testcase or an individual test measure.

They consist of a several parts, some of which may be omitted in some circumstances

The parts are:

- testsuite name
- testset name
- testcase name
- measure name

Legal characters for these parts are letters, numbers, and underscores. Only testset names may include a period (“.”), as that is used as the separator between constituent parts of the identifier.

testcase identifiers should be consistent from run-to-run of a test, and should be globally unique.

test identifiers may be in fully-qualified form, or in shortened form - missing some parts. The following rules are used to convert between from shortened forms to fully-qualified forms.

If the testsuite name is missing, then the base name of the test is used.

- e.g. Functional.jpeg has a default testsuite name of “jpeg”

If the testset name is missing, then the testset name is “default”.

A test id may refer to one of two different items:

- a testcase id
- a measure id

A fully qualified test identifier consists of a testsuite name, testset name and a testcase name. Shortened names may be used, in which case default values will be used for some parts, as follows:

If a result id has only 1 part, it is the testcase name. The testset name is considered to be *default*, and the testsuite name is the base name of the test.

That is, for the fuego test Functional.jpeg, a shortened tguid of *test4*, the fully qualified name would be:

- jpeg.default.test4

If a result id has 2 parts, then the first part is the testset name and the second is the testcase name, and the testsuite name is the base name of the test.

measure id

A measure identifier consists of a testsuite id, testset id, testcase id and measure name.

A shortened measure id may not consist of less than 2 parts. If it only has 2 parts, the first part is the testcase id, and the second part is the measure name. In all cases the last part of the name is the measure name, the second-to-last part of the name is the testcase name.

If there are three parts, the first part is the testset name.

7.23.9 Test variable names

Test variable names are defined in the board file, and by the user via ‘ftc set-var’. Also they are generated from variables declared in spec files. They consist of all upper-case, using only letters and underscores.

Some test variable prefixes and suffixes are used in a consistent way.

Dependency check variables

The following is the preferred format for variables used in dependency checking code:

- **PROGRAM_FOO** - require program ‘foo’ on target. The program name is upper-cased, punctuation or spaces are replaced with ‘_’, and the name is prefixed with ‘PROGRAM_’. The value of variable is full path on target where program resides.
 - ex: PROGRAM_BC=/usr/bin/bc
- **HEADER_FOO** - require header file ‘foo’ in SDK. The header filename is stripped of its suffix (I don’t know if that’s a good idea or not), upper-cased, punctuation or spaces are replaced with ‘_’, and the name is prefixed with ‘HEADER_’. The value of variable is the full path in the SDK of the header file:
 - ex: HEADER_FOO=/opt/poky2.1.2/sysroots/x86_64-pokysdk- linux/usr/include/foo.h
- **SDK_LIB_FOO** - require ‘foo’ library in SDK. The library filename is stripped of the ‘lib’ prefix and .so suffix, upper-cased, punctuation and spaces are replaced with ‘_’, and the name is prefixed with ‘SDK_LIB_’. The value of the variable is the full path in the SDK of the library.
 - ex: SDK_LIB_FOO=/opt/poky2.1.2/sysroots/x86_64-pokysdk- linux/usr/lib/libfoo.so
 - Note that in case a static library is required (.a), then the variable name should include that suffix:
 - ex: SDK_LIB_FOO_A=/opt/poky1.2.1/sysroots/x86_64-pokysdk- linux/usr/lib/libfoo.a
- **TARGET_LIB_FOO** - require ‘foo’ library on target. The library filename is stripped of the ‘lib’ prefix and .so suffix (not sure this is a good idea, as we potentially lose a library version requirement), upper-cased, punctuation and spaces are replaced with ‘_’, and the name is prefixed with ‘TARGET_LIB_’. The value of the variable is the full path of the library on the target board.
 - ex: TARGET_LIB_FOO=/usr/lib/libfoo.so

7.24 Artwork

This page has artwork (logos, photos and images) for use in Fuego presentations and documents.

7.24.1 Logos

- fuego-logo.svg (SVG, high-res)



- fuego-logo.png (png, 660x781 w/ transparent background)



- fuego-flame-only.png: (png, 490x617 w/ transparent background)



- fuego-logo-from-Daniel.svg (SVG, high-res)

Please see this image in the Fuego wiki at: <http://fuegotest.org/files/fuego-logo-from-Daniel.svg>
(Sphinx cannot include this image in PDF output)

7.24.2 Banners

- Fuego Jamboree: (png, 567x120)



- firestrip: (jpg, 1710x282)



7.24.3 images

- vertical fire (public domain, png, 687x2712)



- horizontal fire (public domain, jpg, 615x318)



- rectangle fire (public domain, jpg, 3840x2160)



- rectangle fire with Fuego (png, 1431x1056)



7.24.4 Photos

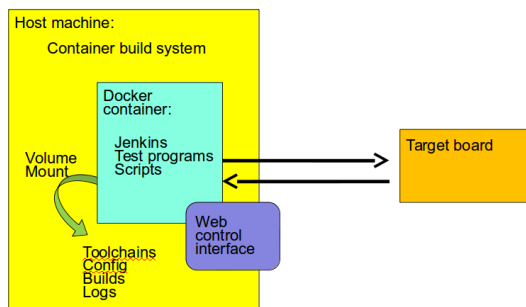
- Tim Bird presenting Introduction to Fuego at ELC 2016 (Youtube video framegrab) (png, 370x370)



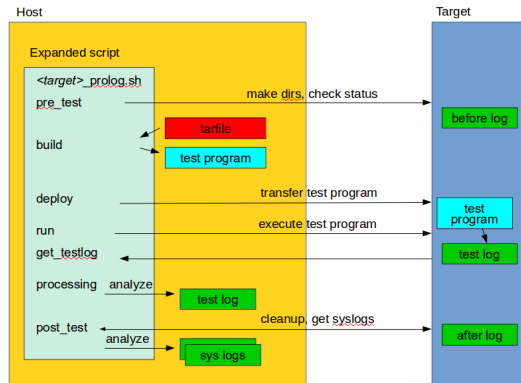
7.24.5 Diagrams

- Fuego architecture (png, 811x566)

Architecture Diagram



- Fuego test phases (png,) {{BR}}



7.24.6 Presentation templates

There are templates for making Fuego presentations, on the Fuego wiki at: http://fuegotest.org/wiki/Presentation_Templates

7.25 Glossary

Here is a glossary of terms used in this wiki:

Here is a short table that relates a few Jenkins terms to Fuego terms:

Jenkins term	Fuego term	Description
slave	“none”	this is a long-running jenkins process, that executes jobs. It is usually (?) assigned to a particular node
node	board	item being tested (Fuego defines Jenkins node for each board in the system)
job	test	a collection of information needed to perform a single test
“none”	request	a request to run a particular test on a board
build	run(or ‘test run’)	the results from executing the job or test
“none”	plan	the plan has the list of tests and how to run them (which variation, or ‘spec’ to use)
“none”	spec	the spec indicates a particular variation of a test

7.25.1 B

base test script

See test script.

benchmark

A type of test where one or more numeric metrics indicates the status of the test. See Benchmark parser notes for more information about processing these metrics.

binary package

A tarfile containing the materials that would normally be deployed to the board for execution.

board file

A file that describes (using environment variables) the attributes of a target board. This has the extension `.board` and is kept in the directory `/fuego-ro/boards`.

build

In Jenkins terminology, a “build” is an execution of a test, and the resulting artifacts associated with it. In Fuego, this is also referred to as a test “run”.

7.25.2 C

console log

The full output of execution of a test from Jenkins. See Log files for details.

criteria

This is an expression which indicates whether how a test result should be interpreted. This is also referred to as a “pass criteria”. Criteria files are stored in `criteria.json` files. See [criteria.json](#) for details.

7.25.3 D

devlog

The developer log for a test. See Log files for details.

Device

The name of a target board in the Fuego system.

device under test

In test terminology, this refers to the item being tested. In Fuego, this may also be called the “Device”, “board”, “node”, or `target`

distribution

This refers to a set of software that is running on a Linux machine. Example “distributions” are Debian, Angstrom or Android. The distribution defines file locations, libraries, utilities and several important facilities and services of the machine (such as the init process, and the logger).

7.25.4 F

functional test

A type of test that returns a single pass/fail result, indicating whether the device under test. It may include lots of sub-tests.

7.25.5 J

Jenkins

An advanced continuous integration system, used as the default front-end for the Fuego test framework. see Jenkins

job

In Jenkins terminology, a job is a test

7.25.6 L

log file

Several log files are created during execution of a test. For details about all the different log files, see Log files.

7.25.7 M

metric

A numeric value measured by a benchmark test as the result of the test. This is compared against a threshold value to determine if the test passed or failed. See Benchmark parser notes

7.25.8 O

overlay

This is a set of variables and functions stored in a fuegoclass file, which are used to customize test execution for a particular board. See Overlay Generation for details.

ovgen.py

Program to collect “overlay” data from various scripts and data files, and produce the final test script to run. see Overlay Generation.

7.25.9 P

package

See test package.

parsed log

The test log file after it has been filtered by log_compare. See Log files for details.

parser.py

A python program, included with each Benchmark test, to scan the test log for benchmark metrics, check each against a reference threshold, and produce a plot.png file for the test. See [parser.py](#) and Benchmark parser notes for more information.

provision

To provision a board is to install the system software on it. Some board control systems re-provision a board for every test. In general, Fuego runs a series of tests with a single system software installation.

7.25.10 R

reference log

This file (called “reference.log”) defines the regression threshold (and operation) for each metric of a benchmark test. See [reference.log](#) and [Benchmark parser notes](#)

run

See [test run](#).

7.25.11 S

spec variable

A test variable that comes from a spec file. See [Test variables](#)

stored variable

A test variable that is stored in a read/write file, and can be updated manually or programmatically. See [Test variables](#)

syslog

The system log for a test. This is the system log collected during execution of a test. See [Log files](#) for details.

7.25.12 T

test

This is a collection of scripts, jenkins configuration, source code, and data files used to validate some aspect of the device under test. See [Fuego Object Details](#) for more information.

test log

This is the log output from the actual test program on the target. There are multiple logs created during the execution of a test, and some might casually also be called “test logs”. However, in this documentation, the term “test log” should be used only to refer to the test program output. See [Log files](#) for details.

test package

This is a packaged version of a test, including all the materials needed to execute the test on another host. See [Test package system](#)

test phases

Different phases of test execution defined by Fuego: `pre_test`, `build`, `deploy`, `test_run`, `get_testlog`, `test_processing`, `post_test`. For a description of phases see: [Fuego test phases](#)

test program

A program that runs on the target to execute a test and output the results. This can be a compiled program or a shell script (in which case the build step is empty)

test run

This is a single instance of a test execution, containing logs and other information about the run. This is referred to in Jenkins as a ‘build’.

test script

The shell script that interfaces between the Fuego core system and a test program. This is a small script associated with each test. It is called `fuego_test.sh`, and it provides a set of test functions that are executed on the host (in the container) when a test is run.

The script declares a tarfile, and functions to build, deploy and run the test. The test script runs on the host. This is also called the ‘base test script’. For details about the environment that a script runs in or the functions it may call, see [Variables](#), [Core interfaces](#), and Test Script APIs.

`test variable`

This is the name of a variable available to the a test during it’s execution. See [Test variables](#).

`TOOLCHAIN`

Defines the toolchain or SDK for the device. This is used to select a set of environment variables to define and configure the toolchain for building programs for the intended test target.

`tools.sh`

File containing the definition of toolchain variables for the different platforms installed in the container (and supported by the test environment) See `tools.sh` for details.

7.25.13 V

`variable`

See `test variable`

7.26 Raspberry Pi Fuego Setup

This is a list of instructions for setting up a Raspberry Pi board for use with Fuego. These instructions will help you set up the ssh server, used by Fuego to communicate with the board, and the test directory on the machine, that Fuego will use to store programs and files during a test.

These instructions and the screen shots are for a Raspberry Pi Model 3 B, running “Raspbian 9 (stretch)”.

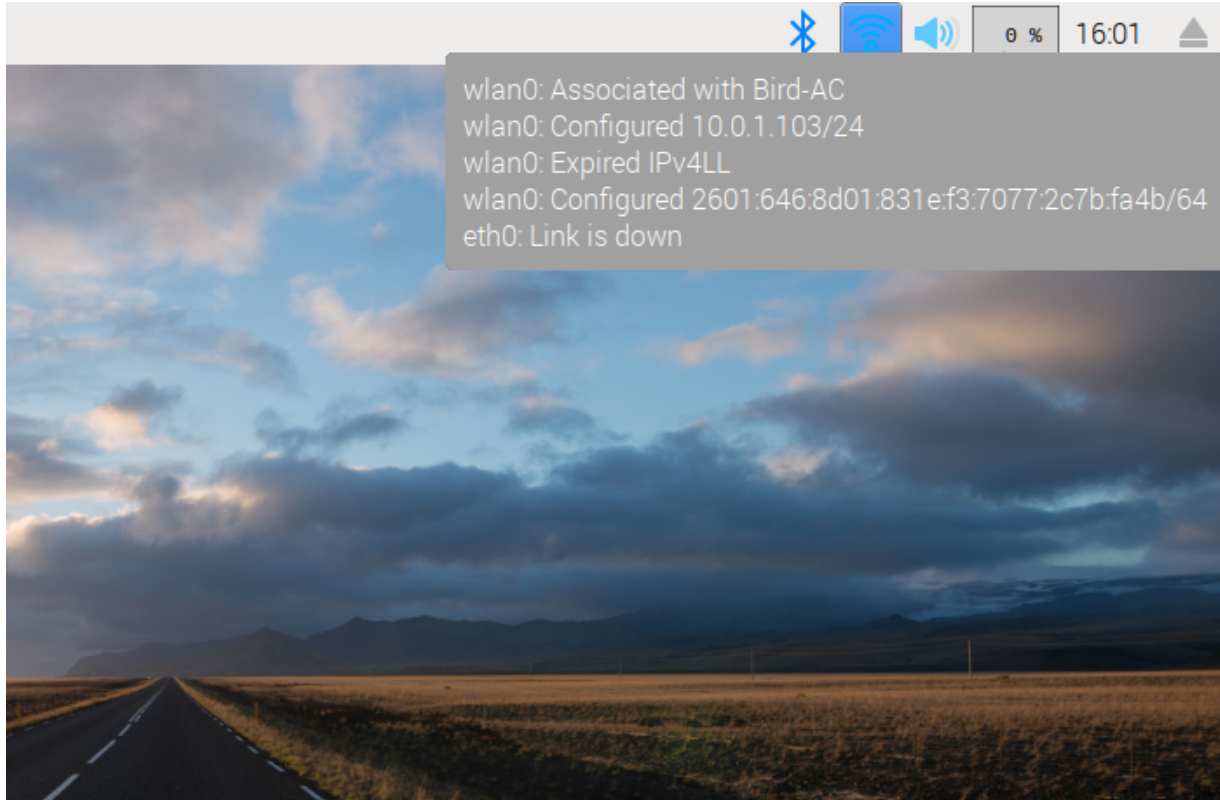
This assumes that the Raspberry Pi is already installed, and that networking is already configured and running.

7.26.1 Obtain your network address

First, determine what your Pi’s network address is. You can see this by using the command ‘ifconfig’ in a terminal window, and checking for the ‘inet’ address.

Or, move your mouse cursor over the network icon in the desktop panel bar. If you leave the mouse there for a second or two, a box will appear showing information about your current network connection.

This is what the network information box looks like (in the upper right corner of this screen shot):



In this case, my network address is 10.0.1.103. Your address might start with 192.168, which is common for home or local networks.

Note this address for use later.

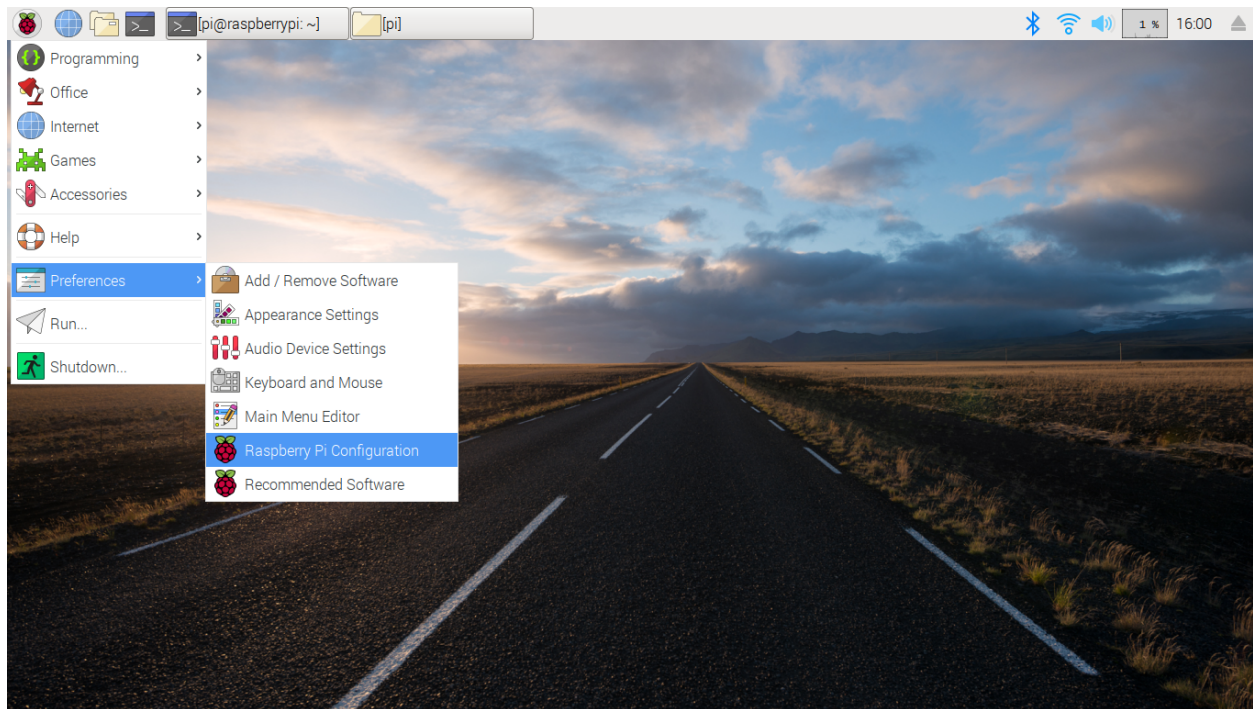
7.26.2 Configure the SSH server

In order for other machines to access the Pi remotely, you need to enable the ssh server.

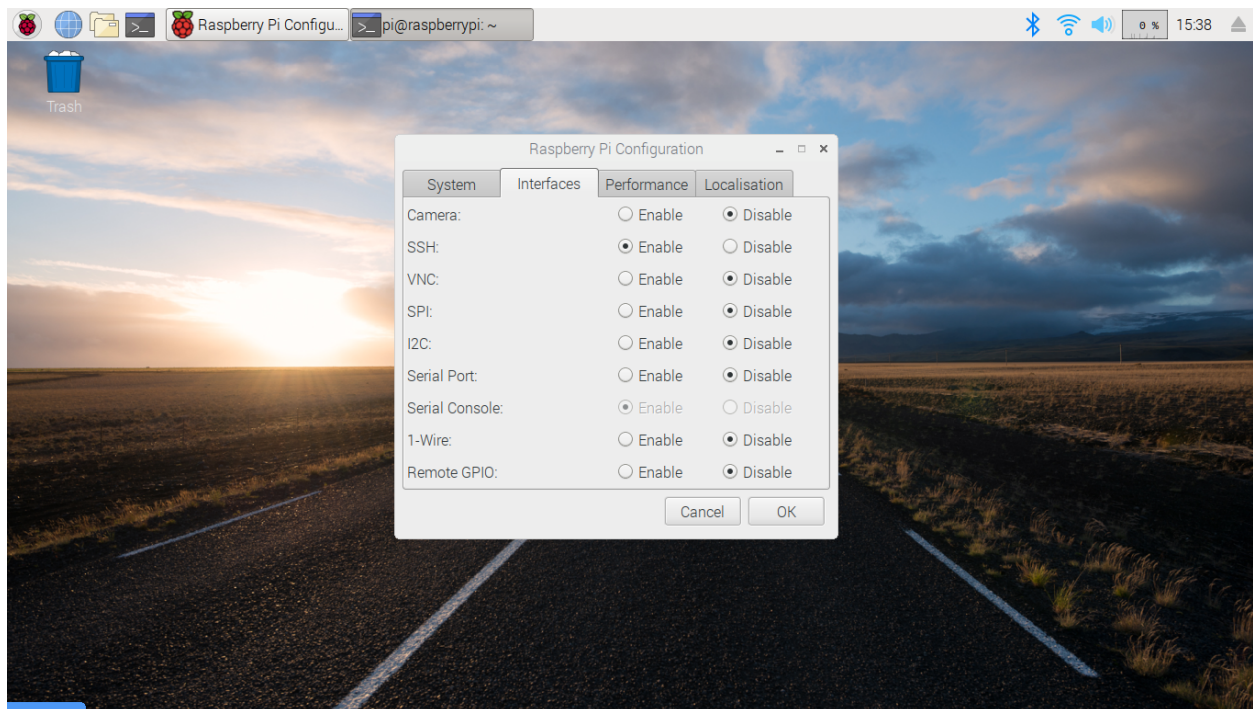
This is done by enabling the SSH interface in the Raspberry Pi Configuration dialog.

To access this dialog, click on the raspberry logo in the upper right corner of the main desktop window. Then click on “Preferences”, then on “Raspberry Pi Configuration”. In the dialog that appears, click on the “Interfaces” tab, and on the list of interfaces click on the “Enable” radio button for the SSH interface.

Here is the menu:



The configuration dialog looks something like this:



Try connecting

Now, close this dialog, and make sure you can access the Pi using SSH from your host machine.

Try the following command, from your host machine:

- `ssh pi@<"your_address">`

You will be asked for the password for the 'pi' user account.

If you successfully log in, you will be at a shell prompt.

Configure sshd for root access (if applicable)

If you intend to execute Fuego tests as root, you should configure the SSH server to permit root login.

This is not recommended on machines that are in production, as it is a significant security risk. However, for test machines it may be acceptable to allow root access over ssh.

To do this, on the Raspberry Pi, with root permissions, edit the file `/etc/ssh/sshd_config` and add the following line:

```
PermitRootLogin yes
```

Note: You need to stop and start the ssh server, or reboot the board, in order to have this new configuration take effect."

7.26.3 Make a test directory

You can use any directory you like for executing tests from Fuego. However, we recommend using the `/home/fuego` directory. These instructions indicate how to create that directory.

If you are using root as your test user account, then create a directory on the Raspberry Pi for test programs and files.

If logged in as the 'pi' account, then switch to root (using something like `'sudo su'`), and type, at a shell prompt:

```
$ mkdir /home/fuego
```

If you do not wish to use the root account for testing, then you should create a 'fuego' account for testing. To do this, use the `'adduser'` program. You will be prompted for some information.

```
$ adduser fuego
```

Answer the questions, including setting the password for this account. Remember the password you select, and use that in the board file when configuring Fuego to access this board.

This will create the directory `/home/fuego`.

The directory `/home/fuego` is what should be used in the board file when configuring Fuego to operate with this board.

7.26.4 Add the board file to Fuego

Now that you have set up the Raspberry Pi board, add the board file to Fuego. Assuming your IP address is 10.0.1.17, and you wish to log in as root, you would create a board file called “rpi.board”, and located at

```
<fuego-top-dir>/fuego-ro/boards/rpi.board
```

with the following contents:

```
inherit "base-board"
include "base-params"

IPADDR="10.0.1.17"
LOGIN="root"
BOARD_TESTDIR="/home/fuego"
PASSWORD="put-the-root-password-here"
TOOLCHAIN="debian-armhf"
TRANSPORT="ssh"
ARCHITECTURE="arm"
FUEGO_TARGET_TMP="/home/fuego"
```

Note: Of course, use the correct root password for your board

7.26.5 Add the toolchain to Fuego

The Raspberry Pi board is an ARM 32-bit platform.

Add the ‘debian-armhf’ toolchain to the Fuego docker container, using the helper script in the fuego-ro/toolchains directory.

Inside the Fuego container, run:

```
$ /fuego-ro/toolchains/install_cross_toolchain.sh armhf
```

7.26.6 Add a node and jobs for the board

Inside the Fuego container, run:

```
$ ftc add-node -b rpi
```

Add the tests you want to run, as Jenkins jobs. You should always add the “fuego_board_check” test, as a way to automatically determine that status of a board.

Inside the Fuego container, run:

```
$ ftc add-job -b rpi -t Functional.fuego_board_check
```

An easy way to populate Jenkins with a set of tests is to install a batch test.

Install the “smoketest” batch test, as follows:

Inside the Fuego container, run:


```
$ ftc add-jobs -b rpi -t Functional.batch_smoketest
```

7.26.7 Run a board check

To see if everything is set up correctly, execute the test: `Functional.fuego_board_check`.

In the Jenkins interface, select “`rpi.default.Functional.fuego_board_check`” and select the menu item “Build Now” on the left hand side of the screen.

Wait a few moments for the test to complete. when the test completes, check the log for the test by clicking on the link to the ‘testlog’.

7.27 Using the qemuarm target

Here are some quick instructions for using the qemu-arm “board” that is preinstalled in fuego.

Fuego does not ship with a qemuarm image in the repository, but assumes that you have built one with the Yocto Project.

If you don’t have one lying around, you will need to build one. Then you should follow the other steps on this page to configure it to run with Fuego.

7.27.1 Build a qemuarm image

Here are some quick steps for building a qemuarm image using the Yocto Project: (See the [Project Quick Start](#), for more information)

Note that these steps are for Ubuntu.

- Make sure you have required packages for building the software

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo
```

```
gcc-multilib build-essential chrpath socat libssl1.2-dev xterm
```

- Install the qemu software

```
$ sudo apt-get install qemu-user
```

- Download the latest stable release of the Yocto Project

```
$ git clone git://git.yoctoproject.org/poky
```

- Configure Yocto Project for building the qemuarm target

```
$ cd poky
$ source oe-init-build-env build-qemuarm build-qemuarm
$ edit conf/local.conf
```

- Under the comment about “Machine Selection”, uncomment the line
‘`MACHINE ?= “qemuarm”`’
- Build a minimal image (this will take a while)

```
$ bitbake core-image-minimal
```

7.27.2 Running the qemuarm image

You can run the emulator, using the image you just built:

- Run the emulator

```
$ runqemu qemuarm
```

- Find the address and ssh port for the image
 - Inside the image, do `ifconfig eth0`

7.27.3 Test connectivity

From the host, verify that the networking is running:

```
$ ping 192.168.7.2
$ ssh root@192.168.7.2
```

Of course, substitute the correct IP address in the commands above.

Once you know that things are working, directly connecting from the host to the qemuarm image, make sure the correct values are in the `qemu-arm.board` file. You can edit this file on your host in `fuego-ro/boards/qemu-arm.board`

Here are the values you should set:

```
IPADDR="192.168.7.2"
SSH_PORT=22
LOGIN="root"
PASSWORD=""
```

7.27.4 Test building software

It is important to be able to build the test software for the image you are using with qemu.

The toolchain used to compile programs for a board is controlled via the `TOOLCHAIN` variable in the board file. Currently the `qemu-arm.board` file specifies `TOOLCHAIN="debian-armhf"`.

You may need to install the `debian-armhf` toolchain, or your own SDK from your Yocto Project build, into the Fuego container in order to build test programs for the qemuarm emulator. See [Adding a toolchain](#) for information about how to do that.

Try building a simple program, like `hello_world`, as a test for the new system, and see what happens. You can try this by executing the test `Functional.hello_world`. You can do that with this command:

```
$ ftc run-test -b qemu-arm -t hello_world
```


7.28 Variables

This is an index of all the variables used by Fuego:

FIXTHIS - I don't have all the fuego variables documented here yet

See also *Core interfaces*

7.28.1 A

- **ARCHITECTURE** : the processor architecture of the target board
 - Defined in the board file for a target
 - Used by toolchain and build scripts for building the tests
 - * *NOTE: this appears to only be used by iozone.sh*
 - Example value: **arm**
- **ARCH** : architecture used by the toolchain
 - Example value: **arm**
 - Set by tools.sh based on TOOLCHAIN
- **AS** : name of the assembler
 - Set by tools.sh based on TOOLCHAIN
 - Commonly used during the build phase (in the function test_build)

7.28.2 B

- **BAUD** : Baud rate to be used with the serialport
 - Defined in the board file for a target
 - Used by serial transport
 - Example value: **115200**
- **BOARD_TESTDIR** : directory on the target board where test data will be placed
 - Defined in the board file for a target
 - Example value: **/home/fuego**

7.28.3 C

- **CC** : name of the C compiler
 - Set by tools.sh based on TOOLCHAIN
 - Commonly used during the build phase (in the function test_build)
 - Example value: **arm-linux-gnueabi-gcc**
- **CONFIGURE_FLAGS** : flags used with the 'configure' program
 - Set by tools.sh based on TOOLCHAIN

- Commonly used during the build phase (in the function `test_build`)
- **CROSS_COMPILE** : cross-compile prefix used for kernel builds
 - Set by `tools.sh` based on `TOOLCHAIN`
 - Example value: **arm-linux-gnueabihf-**
 - *NOTE: this is often \$PREFIX followed by a single dash*
- **CPP** : name of the C pre-processor
 - Set by `tools.sh` based on `TOOLCHAIN`
- **CXX** : name of the C++ compiler
 - Set by `tools.sh` based on `TOOLCHAIN`
- **CXXCPP** : name of the C++ pre-processor
 - Set by `tools.sh` based on `TOOLCHAIN`

7.28.4 F

- **FUEGO_BUILD_FLAGS** : has special flags used to control builds (for some tests)
 - See **FUEGO_BUILD_FLAGS**
- **FUEGO_CORE** : directory for Fuego core scripts and tests
 - This is defined in Jenkins and Fuego system-level configurations
 - This will always be `/fuego-core` inside the Docker container, but will have a different value outside the container.
 - Example value: **/fuego-core**
- **FUEGO_DEBUG** : controls whether Fuego emits debug information during test execution. This variable is now deprecated in favor of **FUEGO_LOGLEVELS**
 - See **FUEGO_DEBUG**
- **FUEGO_LOGLEVELS** : controls what level of messages Fuego emits during test execution
 - See **FUEGO_LOGLEVELS**
- **FUEGO_RO** : directory for Fuego read-only data
 - This is defined in the Jenkins and Fuego system-level configurations
 - This will always be `/fuego-ro` inside the Docker container, but will have a different value outside the container.
 - Example value: **/fuego-ro**
- **FUEGO_RW** : directory for Fuego read-write data
 - This is defined in Jenkins and Fuego system-level configurations
 - This will always be `/fuego-rw` inside the Docker container, but will have a different value outside the container.
 - Example value: **/fuego-rw**
- **FUEGO_TARGET_TMP** : directory on target to use for syslogs
 - This is defined in the board file for a target board

- This should specify a directory in the board filesystem that is persistent across reboots. This is to override the default temp directory (of `/tmp`), if that directory is erased on a board reboot.
- **FUEGO_TEST_PHASES** : specifies a list of phases to perform during test execution
 - This is usually set by `ftc run-test` based on the ‘-p’ option.
 - This is a space-separated list of strings, from the following possible individual phase strings: **pre_test**, **pre_check**, **build**, **deploy**, **snapshot**, **run**, **post_test**, **processing**, **makepkg**
 - Example value: **pre_test pre_check build deploy snapshot run post_test processing**

7.28.5 G

- **GEN_TESTRES_FILE** : set to the value of **TEST_RES**, when a **BATCH_TESTPLAN** is in effect

7.28.6 I

- **IO_TIME_SERIAL** : Time required for echoing the whole command and response
 - Defined in the board file
 - Used by the transport functions
 - Example value: **0.1**
- **IPADDR** : IP address of the target board
 - Defined in the board file
 - Used by the transport functions
 - Example value: **10.0.1.74**

7.28.7 L

- **LD** : name of the linker
 - Set by `tools.sh` based on **TOOLCHAIN**
 - Example value: **arm-linux-gnueabi-hf-ld**
- **LOGIN** : login account name for the target
 - Defined in the board file for the target
 - Used by the transport functions
 - The account on the target should have sufficient rights to run a variety of tests and perform a variety of operations on the target
 - Example value: **root**

7.28.8 M

- **MAX_BOOT_RETRIES** : Number of times to retry connecting to target during a target_reboot operation.
 - Defined in the board file
 - Example value: **20**
- **MMC_DEV** : device filename for an MMC device on the target
 - Defined in the board file
 - Used by filesystem test specs
 - Example value: **/dev/mmcblk0p2**
- **MMC_MP** : mount point for a filesystem on an MMC device on the target
 - Defined in the board file
 - Used by filesystem test specs
 - Example value: **/mnt/mmc**
- **MOUNT_BLOCKDEV** : device filename for a block device on the target
 - Defined in a filesystem test spec
 - * e.g. in (bonnie, fio, ffsb, iotest, synctest, aiostress, dbench, tiobench).spec
 - Usually references either **MMC_DEV**, **SATA_DEV** or **USB_DEV**, depending on what the test spec indicates to test
 - Example value: **/dev/sda1**
- **MOUNT_POINT** : mount point for a filesystem to be tested on the target
 - Defined in a filesystem test spec
 - * e.g. in (bonnie, fio, ffsb, iotest, synctest, aiostress, dbench, tiobench).spec
 - Usually references either **MMC_MP**, **SATA_MP**, or **USB_MP**, depending on what the test spec indicates to test
 - Example value: **/mnt/sata**

7.28.9 N

- **NODE_NAME** : the name of the board
 - This is set by Jenkins, and is the first part of the Fuego job name

7.28.10 O

- **OF_ROOT** : root of overlay system
 - Example value: **/home/jenkins/overlays/**

7.28.11 P

- **PASSWORD** : password used with to login to the target board
 - Defined in the board file for a target
 - Used by the transport functions
 - It can be the empty string: ""
 - Example value: **mypass**
- **PLATFORM** : name of the target “platform” This is used to identify the toolchain used for building tests. This has been deprecated. Please use **TOOLCHAIN** instead.
- **PREFIX** : toolchain prefix
 - Set by tools.sh based on **TOOLCHAIN**
 - Example value: **arm-linux-gnueabi**
 - *NOTE: see also CROSS_COMPILE*

7.28.12 R

- **REP_DIR** : directory where reports are stored
 - Example value: **/home/jenkins/logs/logruns/**
- **REP_GEN** : report generator program
 - Example value: **/home/jenkins/scripts/loggen/gentextml.py**
- **REP_LOGGEN** : program used to generate report logs?
 - Example value: **/home/jenkins/scripts/loggen/loggen.py**

7.28.13 S

- **SATA_DEV** : device node filename for a SATA device on the target
 - Defined in the board file
 - Used by filesystem tests
 - Example value: **/dev/sda1**
- **SATA_MP** : mount point for a filesystem on a SATA device on the target
 - Used by filesystem tests
 - Example value: **/mnt/sata**
- **SRV_IP** : IP address of server machine (host machine where fuego runs)
 - Defined in base-board.fuegoclass
 - * Obtained dynamically using the **ip** command
 - Can be defined in a board file for a target, using an **override** command
 - Used by networking tests (NetPIPE, iperf, netperf)
 - Example value: **10.0.1.1**
- **SSH_PORT** : port to use for ssh connections on the target

- Defined in the board file for the target
- The default port for sshd is 22
- Example value: **22**
- **SERIAL** : port to use for serial connections on the target
 - Defined in the board file for the target
 - The device file name as detected in Docker container
 - Example value: **ttyACM0**

7.28.14 T

- **TESTLOG** : full path to log for a particular test
 - Example value: **/home/jenkins/logs/Functional.bzip2/ testlogs/bbb.2016-06-24_18-12-53.2.log**
- **TEST_RES** : full path to JSON results file for a test
 - Example value: **/home/jenkins/logs/Functional.bzip2/testlogs/ bbb.2016-06-24_18-12-53.2.res.json**
- **TESTDIR** : name of the directory for a particular test
 - This is just the directory name, not the full path (see \$TEST_HOME)
 - This is also used as the reference parse log prefix
 - Example value: **Functional.bzip2**
- **TEST_HOME** : full path to the root of the test directory
 - Example value: **/fuego-core/tests/Functional.bzip2**
- **TOOLCHAIN** : name of the toolchain used to build test programs for a board.
 - Defined in the board file
 - Used in **tools.sh**
 - Example value: **debian-armhf**
 - *NOTE: this replaced 'PLATFORM', used in earlier versions of Fuego*
- **TRANSPORT** : type of connection between the host system and the target system
 - Defined in the board file for the target
 - possible values: **ssh, serial, ttc, ssh2serial, local**
 - * Others anticipated are: **adb, lava**
 - Used by the transport functions
 - Example value: **ssh**
- **TTC_TARGET** : target name used with **ttc** command
 - Defined in the board file for the target
 - Used by the transport functions, for the **ttc** transport only
 - Example value: **beaglebone**

7.28.15 U

- **USB_DEV** : device filename for an block device provided by a USB device on the target
 - Defined in the board file
 - Used by filesystem test specs
 - Example value: **/dev/sdb1**
- **USB_MP** : mount point for a filesystem on an USB device on the target
 - Defined in the board file
 - Used by filesystem test specs
 - Example value: **/mnt/usb**

7.28.16 UNDOCUMENTED (YET)

- **TRIPLET**
- **LTP_OPEN_POSIX_SUBTEST_COUNT_POS**
 - Defined in board file for a target
- **LTP_OPEN_POSIX_SUBTEST_COUNT_NEG**
 - Defined in board file for a target
- **EXPAT_SUBTEST_COUNT_POS**
 - Defined in board file for a target
- **EXPAT_SUBTEST_COUNT_NEG**
 - Defined in board file for a target
- **OF_ROOT**
- **OF_CLASSDIR**
- **OF_DEFAULT_SPECDIR**
- **OF_OVFILES**
- **OF_CLASSDIR_ARGS**
- **OF_TESTPLAN_ARGS**
- **OF_SPECDIR_ARGS**
- **OF_OUTPUT_FILE**
- **OF_OUTPUT_FILE_ARGS**
- **OF_DISTRIB_FILE**
- **OF_OVGEN**
- **OF_BOARD_FILE**
- **BATCH_TESTPLAN**
- **OF_TESTPLAN**
- **OF_TESTPLAN_ARGS**

- `OF_OVFILES_ARGS`

7.29 Dynamic Variables

“Dynamic variables” in Fuego are variables that can be passed to a test on the command line, and used to customize the operation of a test, for a particular test run.

In general testing nomenclature this is referred to as test parameterization.

The purpose of dynamic variables is to support “variant” testing, where a script can loop over a test multiple times, changing the variable to different values.

In Fuego, during test execution dynamic variable names are expanded to full variables names that are prefixed with the name of the test. A dynamic variable overrides a spec variable of the same name.

Here is an example of using dynamic variables:

```
$ ftc run-test -b beaglebone -t Benchmark.Dhrystone --dynamic-vars "LOOPS=100000000"
```

This would override the default value for `BENCHMARK_DHRYSTONE_LOOPS`, setting it to 100000000 (100 million) for this run. Normally, the default spec for `Benchmark.Dhrystones` specifies a value of 10000000 (10 million) for `LOOPS`.

This feature is intended to be useful for (among other things) doing ‘git bisect’s of a bug, passing a different git commit id for each iteration of the test.

See [Test_variables](#) for more information.

7.29.1 Notes

Note that dynamic vars are added to the runtime `spec.json` file, which is saved in the log directory for the run being executed.

This `spec.json` file is copied from the one specified for the run (usually from the test’s home directory).

If dynamic variables have been defined for a test, then they are listed by name in the run-specific `spec.json` file, as the value of the variable “`dyn_vars`”. The reason for this is to allow someone who reviews the test results later to easily see whether a particular test variable had a value that derived from the spec, or from a dynamic variable. This is important for proper results interpretation.

7.30 Troubleshooting Guide

This page describes problems encountered using Fuego, and their solutions.

Note: for Editors: please put each issue in it’s own page section.

7.30.1 Installation

Problem with default Jenkins port

Fuego has Jenkins default to using port 8090 on the host system. However, if you have something else already running on port 8090, you may wish to change this.

You can change the Jenkins port during installation of Fuego, using an argument to the `install.sh` script. For example, to install Fuego with Jenkins configured to use port 9999, use the following command during installation:

```
$ ./install.sh fuego 9999
```

To change the Jenkins port for an already-built Fuego container, start the container, and inside the container edit the file:

- `/etc/default/jenkins`

Change the line that says: `HTTP_PORT=8090`

Change to port to whatever your like.

Also, check the line that defines `JENKINS_ARGS`. Mine looked like this:

```
JENKINS_ARGS="--webroot=/var/cache/jenkins/war --httpPort=8090 --prefix=/fuego"
```

Change this line to read as follows:

```
JENKINS_ARGS="--webroot=/var/cache/jenkins/war --httpPort=$HTTP_PORT --prefix=/fuego"
```

Then restart Jenkins:

- `$ service jenkins restart`

Problem creating docker file

Make sure you are running on a 64-bit version of the Linux kernel on your host machine.

Problem starting Jenkins after initial container creation

Doug Crawford reported a problem starting Jenkins in the container after his initial build.

```
$ sudo ./docker-create-container.sh
Created JTA container 6a420f901af7847f2afa3100d3fb3852b71bc65f92aecd13a9aefe0823d42b77
$ sudo ./docker-start-container.sh Starting JTA container
6a420f901af7847f2afa3100d3fb3852b71bc65f92aecd13a9aefe0823d42b77
[....] Starting Jenkins Continuous Integration Server: jenkins: System error failed!
[ ok ] Starting OpenBSD Secure Shell server: sshd.
[ ok ] Starting network benchmark server.
```

The error string is `jenkins: System error`

Takuo Kogushi provides the following response:

I had the same issue. I did some search in the net and found it is not a problem of fuego itself. As far as I know there are two workarounds;

- 1) Rebuild and install `libpam` with `--disable-audit` option (in the container) or

- 2) Modify `docker-create-container.sh` to add `--pid="host"` option to docker create command

Here is a patch provided by Koguchi-san:

```
diff --git a/fuego-host-scripts/docker-create-container.sh b/fuego-host-scripts/docker-
↪create-container.sh
index 2ea7961..24663d6 100755
--- a/fuego-host-scripts/docker-create-container.sh
+++ b/fuego-host-scripts/docker-create-container.sh
@@ -7,7 +7,7 @@ while [ -h "$SOURCE" ]; do # resolve $SOURCE until the file is no longer
↪a symli done DIR="$( cd -P "$( dirname "$SOURCE" )" && pwd )"

-CONTAINER_ID=`sudo docker create -it -v $DIR/../userdata:/userdata --net="host" fuego`
+CONTAINER_ID=`sudo docker create -it -v $DIR/../userdata:/userdata --pid="host" --net=
↪"host" fuego`
CONTAINER_ID_FILE="$DIR/../last_fuego_container.id"
echo "Created Fuego container $CONTAINER_ID"
echo $CONTAINER_ID > $DIR/../last_fuego_container.id
```

Actually I have not tried the first one and do not know if there is any side effects for the second.

This may be related to this docker bug: <https://github.com/docker/docker/issues/5899>

7.30.2 General

Timeout executing ssh commands

In some cases, the ssh command used by Fuego takes a very long time to connect. There is a timeout for the ssh commands, specified as 15 seconds in the cogent repository and 30 seconds in the fuegotest repository.

The timeout for ssh commands is specified in the file

- `/fuego-core/scripts/overlays/base/base-params.fuegoclass`

You can change `ConnectTimeout` to something longer by editing the file.

```
FIXTHIS - make ConnectTimeout for ssh connections a board-level test variable
```

ssh commands taking a long time

Sometimes, even if the command does not time, the SSH operations on the target take a very long time for each operation.

The symptom is that when you are watching the console output for a test, the test stops at the point of each SSH connection to the target.

One cause of long ssh connection times can be that the target ssh server (sshd) is configured to do DNS lookups on each inbound connection.

To turn this off, on the target, edit the file:

- `/etc/ssh/sshd_config`

and add the line:

```
UseDNS no
```

This line can be added anywhere in the file, but I recommend adding it right after the UsePrivilegeSeparation line (if that's there).

7.31 OSS Test Vision

This page describes aspects of vision for the Fuego project, along with some ideas for implementing specific ideas related to this vision.

7.31.1 Overview of concepts

1. Decentralized testing
2. Automated selection of tests based on test or platform attributes
3. Standardized definition of test attributes and dependencies
4. Way to connect developers with relevant test hardware
5. Test Store (a public repository of available tests)
6. Way to share test-related information (useful parameters, results interpretation)
7. Standards for test packaging
8. Provide incentives for test activities

Letter to ksummit discuss

Here's an e-mail Tim sent to the ksummit-discuss list in October, 2016

I have some ideas on Open Source testing that I'd like to throw out there for discussion. Some of these I have been stewing on for a while, while some came to mind after talking to people at recent conference events.

Sorry - this is going to be long...

First, it would be nice to increase the amount of testing we do, by having more test automation. (ok, that's a no-brainer). Recently there has been a trend towards more centralized testing facilities, like the zero-day stuff or board farms used by *KernelCI*. That makes sense, as this requires specialized hardware, setup, or skills to operate certain kinds of test environments. As one example, an automated test of kernel boot requires automated control of power to a board or platform, which is not very common among kernel developers.

A centralized test facility has the expertise and hardware to add new test nodes relatively cheaply. They can do this more quickly and much less expensively than the first such node by an individual new to testing.

However, I think to make great strides in test quantity and coverage, it's important to focus on ease of use for

(continues on next page)

(continued from previous page)

individual test nodes. My vision would be to have tens of thousands of individual test nodes running automated tests on thousands of different hardware platforms and configurations and workloads.

The kernel selftest project is a step in the right direction for this, because it allows any kernel developer to easily (in theory) run automated unit tests for the kernel. However, this is still a manual process. I'd like to see improved standards and infrastructure for automating tests.

It turns out there are lots of manual steps in the testing and bug-fixing process with the kernel (and other Linux-related software). It would be nice if a new system allowed us to capture manual steps, and over time convert them to automation.

Here are some problems with the manual process that I think need addressing:

1) How does an individual know what tests are valid for their platform? Currently, this is a manual decision. In a world with thousands or tens of thousands of tests, this will be very difficult.

We need to have automated mechanisms to indicate which tests are relevant for a platform. Test definitions should include a description of the hardware they need, or the test setup they need. For example, it would be nice to have tests indicate that they need to be run on a node with USB gadget support, or on a node with the gadget hardware from a particular vendor (e.g. a particular SOC), or with a particular hardware phy (e.g. Synopsis). As another example, if a test requires that the hardware physically reboot, then that should be indicated in the test. If a test requires that a particular button be pressed (and that the button be available to be pressed), it should be listed. Or if the test requires that an external node be available to participate in the test (such as a wifi endpoint, CANbus endpoint, or i2C device) be present, that should be indicated. There should be a way for the test nodes which provide those hardware capabilities, setups, or external resources to identify themselves.

Standards should be developed for how a test node and a test can express these capabilities and requirements. Also, standards need to be developed so that a test can control those external resources to participate in tests. Right now each test framework handles this in its own way (if it provides support for it at all).

I heard of a neat setup at one company where the video output from a system was captured by another video system,

(continues on next page)

(continued from previous page)

and the results analyzed automatically. This type of test setup currently requires an enormous investment of expertise, and possibly specialized hardware. Once such a setup is performed in a few locations, it makes much more sense to direct tests that need such facilities to those locations, than it does to try to spread the expertise to lots of different individuals (although that certainly has value also).

For a first pass, I think the kernel CONFIG variables needed by a test should be indicated, and they could be compared with the config for the device under test. This would be a start on the expression of the dependencies between a test and the features of the test node.

2) How do you connect people who are interested in a particular test with a node that can perform that test?

My proposal here is simple - for every subsystem of the kernel, put a list of test nodes in the MAINTAINERS file, to indicate nodes that are available to test that subsystem. Tests can be scheduled to run on those nodes, either whenever new patches are received for that sub-system, or when a bug is encountered and developers for that subsystem want to investigate it by writing a new test. Tests or data collection instructions that are now provided manually would be converted to formal test definitions, and added to a growing body of tests. This should help people re-use test operations that are common. Capturing test operations that are done manually into a script would need to be very easy (possibly itself automated), and it would need to be easy to publish the new test for others to use.

Basically, in the future, it would be nice if when a person reported a bug, instead of the maintainer manually walking someone through the steps to identify the bug and track down the problem, they could point the user at an existing test that the user could easily run.

I imagine a kind of "test app store", where a tester can select from thousands of tests according to their interest. Also, people could rate the tests, and maintainers could point people to tests that are helpful to solve specific problems.

3) How does an individual know how to execute a test and how to interpret the results?

For many features or sub-systems, there are existing tools (e.g bonnie for filesystem tests, netperf for networking tests, or cyclictest for realtime), but these tools have a variety of options for testing different aspects of a

(continues on next page)

(continued from previous page)

problem or for dealing with different configurations or setups. Online you can find tutorials for running each of these, and for helping people interpret the results. A new test system should take care of running these tools with the proper command line arguments for different test aspects, and for different test targets ('device-under-test's).

For example, when someone figures out a set of useful arguments to `cyclictest` for testing realtime on a beaglebone board, they should be able to easily capture those arguments to allow another developer using the same board to easily re-use those test parameters, and interpret the `cyclictest` results, in an automated fashion. Basically we want to automate the process of finding out "what options do I use for this test on this board, and what the heck number am I supposed to look at in this output, and what should its value be?".

Another issue is with interpretation of test results from large test suites. One notorious example of this is LTP. It produces thousands of results, and almost always produces failures or results that can be safely ignored on a particular board or in a particular environment. It requires a large amount of manual evaluation and expertise to determine which items to pay attention to from LTP. It would be nice to be able to capture this evaluation, and share it with others with either the same board, or the same test environment, to allow them to avoid duplicating this work.

Of course, this should not be used to gloss over bugs in LTP or bugs that LTP is reporting correctly and actually need to be paid attention to.

4) How should this test collateral be expressed, and how should it be collected, stored, shared and re-used?

There are a multitude of test frameworks available. I am proposing that as a community we develop standards for test packaging which include this type of information (test dependencies, test parameters, results interpretation). I don't know all the details yet. For this reason I am coming to the community see how others are solving these problems and to get ideas for how to solve them in a way that would be useful for multiple frameworks. I'm personally working on the Fuego test framework - see <http://fuegotest.org/wiki>, but I'd like to create something that could be used with any test framework.

5) How to trust test collateral from other sources (tests, interpretation)

(continues on next page)

(continued from previous page)

One issue which arises with this type of sharing (or with any type of sharing) is how to trust the materials involved. If a user puts up a node with their own hardware, and trusts the test framework to automatically download and execute a never-before-seen test, this creates a security and trust issue. I believe this will require the same types of authentication and trust mechanisms (e.g. signing, validation and trust relationships) that we use to manage code in the kernel.

I think this is more important than it sounds. I think the real value of this system will come when tens of thousands of nodes are running tests where the system owners can largely ignore the operation of the system, and instead the test scheduling and priorities can be driven by the needs of developers and maintainers who the test node owners have never interacted with.

Finally,

6) What is the motivation for someone to run a test on their hardware?

Well, there's an obvious benefit to executing a test if you are personally interested in the result. However, I think the benefit of running an enormous test system needs to be de-coupled from that immediate direct benefit. I think we should look at this the same way we look at other crowd-sourced initiatives, like Wikipedia. While there is some small benefit for someone producing an individual page edit, we need to move beyond that to the benefit to the community of the cumulative effort.

I think that if we want tens of thousands of people to run tests, then we need to increase the cost/benefit ratio for the system. First, you need to reduce the cost so that it is very cheap, in all of [time|money|expertise| ongoing attention], to set up and maintain a test node. Second, there needs to be a real benefit that people can measure from the cumulative effect of participating in the system. I think it would be valuable to report bugs found and fixed by the system as a whole, and possibly to attribute positive results to the output provided by individual nodes. (Maybe you could 'game-ify' the operation of test nodes.)

Well, if you are still reading by now, I appreciate it. I have more ideas, including more details for how such a system might work, and what types of things it could accomplish. But I'll save that for smaller groups who might be more directly interested in this topic.

To get started, I will begin working on a prototype of a test packaging system that includes some of the ideas

(continues on next page)

(continued from previous page)

mentioned here: inclusion of test collateral, and package validation. I would also like to schedule a "test summit" of some kind (maybe associated with ELC or Linaro Connect, or some other event), to discuss standards in the area I propose.

I welcome any response to these ideas. I plan to discuss them at the upcoming test framework mini-jamboree in Tokyo next week, and at Plumbers (particularly during the 'testing and fuzzing' session) the week following. But feel free to respond to this e-mail as well.

Thanks.

-- Tim Bird

7.31.2 Ideas related to the vision

Capturing tests easily

- It should be easy to capture a command line sequence, and test the results
- It might be nice to do an automated capture of command output, and format the output into a `clitest` file that can be used as a here document inside a Fuego test script.

7.31.3 Test definition or attributes

- Do test definitions need to be board-specific
- Elements of test definition:
 - Test dependencies:
 - * Kernel config values needed
 - * Kernel features needed:
 - proc filesystem
 - sys filesystem
 - trace filesystem
 - * Test hardware needed
 - * Test node setup features
 - ability to reboot the board
 - ability to soft-reset the board
 - ability to install a new kernel
 - * Presence of certain programs on target
 - bc
 - top, ps, /bin/sh, bash?
- Fuego already has:

- CAPABILITIES?
- pn and reference logs
- positive and negative result counts (specific to board)
- test specs indicate parameters for the test
- test plans indicate different profiles (method to match test to test environment - e.g. filesystem test with type of filesystem hardware)

7.31.4 Test app store

It would be nice to have an “Test Store”, similar to an “App Store”, where tests can be made publicly available, and browsed and installed by test developers, based on their needs.

Here are some of the items needed for this project:

- Need a repository where tests can be downloaded
 - similar to a Jenkins plugin repository
 - or similar to a Debian package feed
- Need a client for browsing tests, installing tests, updating tests
- It might be possible to store tests in github, and just refer to different tests in different git repositories?
- It would be nice to have test ratings, including user feedback on tests
- It would be nice to have test metrics (e.g. how many bugs has the test found)

7.31.5 Authenticating tests

It is important, if you have a public repository of tests, that you introduce an element of trust and authentication to the repository to avoid malicious actions. You may want to have an authority review the test and possibly sign it. An open question would be who would be the trusted authority (Fuego maintainers? This would turn into a bottleneck).

7.31.6 Test system metrics

It is useful for a test system to provide information about the number of bugs that a test system finds and that get fixed in upstream software. Also, a test system will find bugs in test programs and in itself. These should be noted as well.

7.32 Test Specs and Plans

Note: This page describes (among other things) the standalone test plan feature, which is in process of being converted to a new system. The information is still accurate as of November 2020 (and Fuego version 1.5). However, standalone testplans are now deprecated. The testplan system is being refactored and testplan json data is being integrated into a new batch test system introduced in Fuego version 1.5. See [Using Batch Tests](#) for more information.

7.32.1 Introduction

Fuego provides a mechanism to control test execution using something called “test specs” and “test plans”. Together, these allow for customizing the invocation and execution of tests in the system. A test spec lists variables, with their possible values for different test scenarios, and the test plan lists a set of tests, and the set of variable values (the spec) to use for each one.

There are numerous cases where you might want to run the same test program, but in a different configuration (i.e. with different settings), in order to measure or test some different aspect of the system. This is often referred to as test “parameterization”. One of the simplest different types of test settings you might choose is whether to run a quick test or a thorough (long) test. Selecting between quick and long is a high-level concept, and corresponds to the concept of a test plan. The test plan selects different arguments, for the tests that this makes sense for.

For example, the arguments to run a long filesystem stress test, are different than the arguments to run a long network benchmark test. For each of these individual tests, the arguments will be different for different plans.

Another broad category of test difference is what kind of hardware device or media you are running a filesystem test on. For example, you might want to run a filesystem test on a USB-based device, but the results will likely not be comparable with the results for an MMC-based device. This is due to differences in how the devices operate at a hardware layer and how they are accessed by the system. Therefore, depending on what you are trying to measure, you may wish to measure only one or another type of hardware.

The different settings for these different plans (the test variables or test parameters) are stored in the test spec file. Each test in the system has a test spec file, which lists different specifications (or “specs”) that can be incorporated into a plan. The specs list a set of variables for each spec. When a testplan references a particular spec, the variable values for that spec are set by the Fuego overlay generator during the test execution.

In general, test plan files are global and have the names of categories of tests.

Note: Note that a spec mentioned in a test plan may not be available every test. In fact the only spec that is guaranteed to be available in every test is the ‘default’ test spec. It is important for the user to recognize which test specs and plan arguments may be suitably used with which tests.

7.32.2 Test plans

The Fuego “test plan” feature is provided as an aid to organizing testing activities.

There are only a few “real” testplans provided in Fuego (as of early 2019). There is a “default” testplan, which includes a smattering of different tests, and some test plans that allow for selecting between different kinds of hardware devices that provide file systems. Fuego includes a number of different file system tests, and these plans allow customizing each test to run with filesystems on either USB, SATA, or MMC devices.

These test plans allow this selection:

- `testplan_usbstor`
- `testplan_sata`
- `testplan_mmc`

These plans select test specs named: ‘usb’, ‘sata’, and ‘mmc’ respectively.

Fuego also includes some test-specific test plans (for the `Functional.bc` and `Functional.hello_world` tests), but these are there more as examples to show how the test plan and spec system works, than for any real utility.

A test plan is specified by a file in JSON format, that indicates the test plan name, and a list of tests. For each test, it also lists the specs which should be used for that test, when run with this plan. The test plan file should have a descriptive

name starting with `'testplan_'` and ending in the suffix `'.json'`, and the file must be placed in the `overlays/testplans` directory.

Example

The test program from the `hello_world` test allows for selecting whether the test succeeds, always fails, or fails randomly. It does this using a command line argument.

The Fuego system includes test plans that select these different behaviors. These test plan files are named:

- `testplan_default.json`
- `testplan_hello_world_fail.json`
- `testplan_hello_world_random.json`

Here is `testplan_hello_world_random.json`

```
{
  "testPlanName": "testplan_hello_world_random",
  "tests": [
    {
      "testName": "Functional.hello_world",
      "spec": "hello-random"
    }
  ]
}
```

Testplan Reference

A testplan can include several different fields, at the “top” level of the file, and associated with an individual test. These are described on the page: [Testplan_Reference:Testplan Reference](#)

7.32.3 Test Specs

Fuego’s “test spec” system is a mechanism for running Fuego tests in a “parameterized” fashion. That is, you can run the same underlying test program, but with different values for variables that are passed to the test (the test “parameters”, in testing nomenclature). Each ‘spec’ that is defined for a test may also be referred to as a test ‘variant’ - that is, a variation on the basic operation of the test.

Each test in Fuego should have a ‘test spec’ file, which lists different specifications or variants for that test. For each ‘spec’ (or variant), the configuration declares the variables that are recognized by that test, and their values. Every test is required to define a “default” test spec, which is the default set of test variables used when running the test. Note that a test spec is not required to define any test variables, and this is the case for many ‘default’ test specs for tests which have no variants.

The set of variables, and what they contain is highly test-specific. In some cases, a test variable is used to configure different command line options for the test program. In other cases, the variable may be used by `fuego_test.sh` to change how test preparation is done, or to select different hardware devices or file systems for the test to operate on.

The test spec file is in JSON format, and has the name “spec.json”.

The test spec file is placed in the test’s home directory, which is based on the test’s name: `/fuego-core/tests/$TESTNAME/spec.json`

Example

The `Functional.hello_world` test has a test spec that provides options for executing the test normally (the ‘default’ spec), for always failing (the ‘hello-fail’ spec), or for succeeding or failing randomly (the ‘hello-random’ spec)

This test spec file for the ‘hello_world’ test is `fuego-core/tests/Functional.hello_world/spec.json`

Here is the complete spec for this test:

```
{
  "testName": "Functional.hello_world",
  "specs": {
    "hello-fail": {
      "ARG": "-f"
    },
    "hello-random": {
      "ARG": "-r"
    },
    "default": {
      "ARG": ""
    }
  }
}
```

During test execution, the variable `$FUNCTIONAL_HELLO_WORLD_ARG` will be set to one of the three values shown (nothing, ‘-f’ or ‘-r’), depending on which spec is used when the test is run.

In Fuego, the spec to use with a test can be specified multiple different ways:

- as part of the Jenkins job definition
- on the `ftc run-test` command line
- as part of a testplan definition

7.32.4 Variable use during test execution

Variables from the test spec are expanded by the overlay generator during test execution. The variables declared in the test spec files may reference other variables from the environment, such as from the board file, relating to the toolchain, or from the fuego system itself.

The name of the variable is appended to the end of the test name to form the environment variable that is used by the test. The environment variable name is converted to all uppercase. This environment variable can be used in the `fuego_test.sh` as an argument to the test program, or in any other way desired.

Example

In this hello-world example, the program invocation (by `fuego_test.sh`) uses the variable `$FUNCTIONAL_HELLO_WORLD_ARG`. Below is an excerpt from `/fuego-core/tests/Functional.hello_world/fuego_test.sh`.

```
function test_run {
  report "cd $BOARD_TESTDIR/fuego.$TESTDIR; ./hello $FUNCTIONAL_HELLO_WORLD_ARG"
}
```

Note that in the default spec for `hello_world`, the variable ('ARG' in the test spec) is left empty. This means that during execution of this test with `testplan_default`, the program 'hello' is called with no arguments, which will cause it to perform its default operation. The default operation for the 'hello' program is to write "Hello World" and a test result of "SUCCESS", and then exit successfully.

7.32.5 Specifying failure cases

A test spec file can also specify one or more failure cases. These represent string patterns that Fuego scans for in the test log, to detect error conditions indicating that the test failed. The syntax for this is described next.

Example of fail case

The following example of a test spec (from the `Functional.bc` test), shows how to declare an array of failure tests for this test.

There should be a variable named "fail_case" declared in test test spec JSON file, and it should consist of an array of objects, each one specifying a 'fail_regexp' and a 'fail_message', with an optional variable (use_syslog) indicating to search for the item in the system log instead of the test log.

The regular expression is used with `grep` to scan lines in the test log. If a match is found, then the associated message is printed, and the test is aborted.

```
{
  "testName": "Functional.bc",
  "fail_case": [
    {
      "fail_regexp": "some test regexp",
      "fail_message": "some test message"
    },
    {
      "fail_regexp": "Bug",
      "fail_message": "Bug or Oops detected in system log",
      "use_syslog": 1
    }
  ],
  "specs": [
    {
      "name": "default",
      "EXPR": "3+3",
      "RESULT": "6"
    }
  ]
}
```

These variables are turned into environment variables by the overlay generator and are used with the function `fail_check_cases` which is called during the 'post test' phase of the test.

Note that the above items would be turned into the following environment variables internally in the fuego system:

- `FUNCTIONAL_BC_FAIL_CASE_COUNT=2`
- `FUNCTIONAL_BC_FAIL_PATTERN_0="some test regexp"`
- `FUNCTIONAL_BC_FAIL_MESSAGE_0="some test message"`

- FUNCTIONAL_BC_FAIL_PATTERN_1="Bug"
- FUNCTIONAL_BC_FAIL_MESSAGE_1="Bug or Oops detected in system log"
- FUNCTIONAL_BC_FAIL_1_SYSLOG=true

7.32.6 Catalog of current plans

Fuego, as of January 2017, only has a few testplans defined.

Here is the full list:

- testplan_default
- testplan_mmc
- testplan_sata
- testplan_usbstor
- testplan_bc_add
- testplan_bc_mult
- testplan_hello_world_fail
- testplan_hello_world_random

The storage-related testplans (mmc, sata, and usbstor) allow the test spec to configure the appropriate following variables:

- MOUNT_BLOCKDEV
- MOUNT_POINT
- TIMELIMIT
- NPROCS

Both the 'bc' and 'hello_world' testplans are example testplans to demonstrate how the testplan system works.

The 'bc' testplans are for selecting different operations to test in 'bc'. The 'hello_world' testplans are for selecting different results to test in 'hello_world'

7.33 parser.py

7.33.1 PROGRAM

`parser.py`

7.33.2 DESCRIPTION

`parser.py` is a Python program that is used by each test to parse the test log for a test run, check the threshold(s) for success or failure, and store the data used to generate charts.

Each benchmark should include an executable file called ‘`parser.py`’ in its test home directory (`/fuego-core/tests/Benchmark.<testname>`). Functional tests may also provide a `parser.py`, when they return more than a single testcase result from the test. However, this is optional. If a Functional test does not have a `parser.py` script, then a generic one is used (called `generic_parser.py`), that just sets the result for the test based on the return code from the test program and the single result from running `log_compare` in the `test_processing` portion of the test script.

The overall operation of `parser.py` is as follows: `parser.py` reads the test log for the current run and parses it, extracting one or more testcase or measure results. These are stored in a python dictionary which is passed to the results processing engine. Normally the parsing is done by scanning the log using simple regular expressions. However, since this is a python program, an arbitrarily complex parser can be written to extract the result data from the test log.

Outline

The program usually has the following steps:

- Import the parser library
- Specify a search pattern for finding one or more measurements (or testcases) from the test log
- Call the `parse_log` function, to get a list of matches for the search pattern
- Build a dictionary of result values
- Call the `process` function.
 - The `process` function evaluates the results from the test, and determines the overall pass/fail status of a test, based on a `criteria.json` file
 - The `process` function also saves the information to the aggregate results file for this test (`flat_plot_data.txt`), and re-generates the chart data for the test (`flot_chart_data.json`).

Testcase and measure names

The `parser.py` program provides the name for the measures and testcases read from the test log file. It also provides the result values for these items, and passes the parsed data values to the processing routine.

These test names must be consistent in the `parser.py` program, `reference.json` file and the `criteria.json` file.

Please see [Fuego naming rules](#) for rules and guidelines for test names in the Fuego system.

7.33.3 SAMPLES

Here is a sample `parser.py` that does simple processing of a single metric. This is for `Benchmark.Dhrystone`.

Note the two calls to parser library functions: `parse_log()` and `process()`.

```
#!/usr/bin/python

import os, re, sys

sys.path.insert(0, os.environ['FUEGO_CORE'] + '/scripts/parser')
import common as plib
```

(continues on next page)

(continued from previous page)

```
regex_string = "^ (Dhrystones.per.Second:)(\s*)([\\d]{1,8}.?[\\d]{1,3})(.*)$"

measurements = {}
matches = plib.parse_log(regex_string)

if matches:
    measurements['default.Dhrystone'] = [{"name": "Score", "measure" :
↪ float(matches[0][2])}]

sys.exit(plib.process(measurements))
```

7.33.4 ENVIRONMENT and ARGUMENTS

`parser.py` uses the following environment variable:

- FUEGO_CORE

This is used to add `/fuego-core/scripts/parser` to the python system path, for importing the `common.py` module (usually as internal module name `'plib'`).

The parser library expects the following environment variables to be set:

- FUEGO_RW
- FUEGO_RO
- FUEGO_CORE
- NODE_NAME
- TESTDIR
- TESTSPEC
- BUILD_NUMBER
- BUILD_ID
- BUILD_TIMESTAMP
- PLATFORM
- FWVER
- LOGDIR
- FUEGO_START_TIME
- FUEGO_HOST
- Reboot
- Rebuild
- Target_PreCleanup
- WORKSPACE
- JOB_NAME

`parser.py` is called with the following invocation, from `function_processing`:


```
run_python $PYTHON_ARGS $FUEGO_CORE/tests/${TESTDIR}/parser.py
```

7.33.5 SOURCE

Located in `fuego-core/tests/${TESTDIR}/parser.py`.

7.33.6 SEE ALSO

- `parser_func_parse_log`, `parser_func_process`
- `function_processing`, [Parser module API](#), `Benchmark_parser_notes`.

7.34 criteria.json

7.34.1 Introduction

The `criteria.json` file is used to specify the criteria used to determine whether a test has passed or failed. The data and directives in a `criteria.json` file (referred to as the criteria data) allow Fuego to interpret test results and indicate ultimate success or failure of a test.

A test usually produces a number of individual testcase results or measurement values from the execution of the test. For functional tests, the criteria data can include the number of testcases in the test that must “PASS” or that may be allowed to “FAIL”. Or the criteria data can indicate specific testcase results that should be ignored. For benchmark tests, the criteria data includes threshold values for measurements taken by the benchmark, as well as operations (e.g. ‘less than’ or ‘greater than’), to use to determine if the value of a measurement should be interpreted as a “PASS” or a “FAIL”. Fuego uses the results of the test along with the criteria data, to determine the final top-level result of the test.

If no `criteria.json` file is provided, then a default is constructed based on the test results, consisting of the following:

```
{
  'tguid': <test_set_name>
  'max_fail': 0
}
```

Types of tests and pass criteria

A simple functional test runs a short sequence of tests, and if any one of them fails, then the test is reported as a failure. Since this corresponds to the default `criteria.json`, then most simple Functional tests do not need to provide a `criteria.json` file.

A complex functional test (such as LTP or glib) has hundreds or possibly thousands of individual test cases. Such tests often have some number of individual test cases that fail, but which may be safely ignored (either temporarily or permanently). For example, some test cases may fail sporadically due to problems with the test infrastructure or environment. Other tests may fail due to configuration choices for the software on the board. (For example, a choice of kernel config may cause some tests to fail - but this is expected and these fail results should be ignored).

Functional tests that are complex require a `criteria.json` file, to avoid failing the entire test because of individual testcases that should be ignored.

Finally, a Benchmark test is one that produces one or more “measurements”, which are test results with numeric values. In order to determine whether a result indicates a PASS or a FAIL result, Fuego needs to compare the numeric result

with some threshold value. The `criteria.json` file holds the threshold value and operator used for making this comparison.

Different boards, or boards with different software installations or configurations, may require different pass criteria for the same tests. Therefore, the pass criteria are broken out into a separate file that can be adjusted at each test site, and for each board. Ultimately, we would like testers to be able to share their pass criteria, so that each Fuego user does not have to determine these on their own.

7.34.2 Evaluation criteria

The criteria file lists “pass criteria” for test suites, test sets, test cases and measures. A single file may list one or more pass criteria for the test.

The criteria file may include count-based pass criteria, specific testcase lists, and measure reference values (thresholds).

The criteria file specifies the pass criteria for one or more test element results, by specifying the element’s test id (or tguid), and the criterion used to evaluate that element. Some results elements, such as test sets, are aggregates of other elements. For these, the criteria specify attributes of their child elements (like required counts, or listing individual children that must pass or fail).

The criteria file consists of a list of criterion objects (JSON objects), each of which specifies the tguid for the result element of the test, and additional data used to evaluate that element. tguids are generated by Fuego during the processing phase, and consist of statically defined strings unique to each test. You should look at a test’s `run.json` file to see the test element names for a test.

Here are the different operations that can be used for criteria:

- **max_fail** - specifies the maximum number of child elements that can fail, before causing this element to fail
 - by default, every aggregate element must have all it’s children pass, in order for it to pass (corresponding to a ‘max_fail’ of 0)
- **min_pass** - specifies the minimum number of child elements that must pass, in order for this element to pass
- **must_pass_list** - specifies a list of child elements, by name, that must pass for this element to pass
- **fail_ok_list** - specifies a list of child elements, by name, that may fail, without causing this element to fail
- **reference** - specifies a reference value used as a threshold to evaluate where a number value for this element represents pass or fail.
 - the reference object has two sub-attributes:
 - * **value** - the reference value (threshold)
 - * **operator** - the test between the result and the reference value

The operator can be one of the following strings:

- **gt** - result must be greater than the reference value
- **ge** - result must be greater than or equal to the reference value
- **lt** - result must be less than the reference value
- **le** - result must be less than or equal to the reference value
- **eq** - result must equal the reference value
- **ne** - result must not equal the reference value
- **bt** - result is between two reference values (or equal to one of them)

In case the reference object has an operator of 'bt', the 'value' field should have a string consisting of two numbers separated by a ','. For example, to indicate that the result value should be between 4 and 5, the 'value' field should have the string "4,5". Note that the comparison for 'between' also succeeds for equality. So in the example case of a reference value of "4,5", the test would pass if the test result was exactly 4, or exactly 5, or any number between 4 and 5.

Note: The equality and inequality operators ('eq' and 'ne') are less likely to be useful for numerical evaluations of most benchmark measures, but are provided for completeness. These are useful if a test reports numerical results from within a small set of numbers (like 0 and 1).

7.34.3 Customizing the criteria.json file for a board

A Fuego user can customize the pass criteria for a board, by making a copy of the `criteria.json` file, manually editing the contents, and putting it in a specific directory with a specific filename, so Fuego can find it.

Using an environment variable

A Fuego user can specify their own path to the criteria file to use for a test using the environment variable `FUEGO_CRITERIA_JSON_PATH`. This can be set in the environment variables block in the Jenkins job for a test, if running the Fuego test from Jenkins, or in the shell environment prior to running a Fuego test using 'ftc'.

For example, the user could do the following:

- `$ export FUEGO_CRITERIA_JSON_PATH=/tmp/my-criteria.json`
- `$ ftc run-test -b board1 -t Functional.foo`

Using a board-specific directory

More commonly, a user can specify a board-specific criteria file, by placing the file under either `/fuego-rw/boards` or `/fuego-ro/boards`

When Fuego does test evaluation, it searches for the the criteria file to use, by looking for the following files in the indicated order:

- `$FUEGO_CRITERIA_JSON_PATH`
- `/fuego-ro/boards/{board}-{testname}-criteria.json`
- `/fuego-rw/boards/{board}-{testname}-criteria.json`
- `/fuego-core/tests/{testname}/criteria.json`

As an example, a user could customize the criteria file as follows:

- `$ cp /fuego-core/tests/Benchmark.Dhrystone/criteria.json /fuego-rw/boards/board1-Benchmark.Dhrystone-criteria.json`
- `$ edit /fuego-rw/boards/board1-Benchmark.Dhrystone-criteria.json`
 - Alter the reference value for the tguid 'default.Dhrystone.Score' to reflect a value appropriate for their board ('board1' in this example)
- (execute the job 'board1.default.Benchmark.Dhrystone' in Jenkins)
 - Fuego will use the criteria file for board1 in `/fuego-rw` instead of the default `criteria.json` file in the test's home directory

7.34.4 Examples

Here are some example `criteria.json` files:

Benchmark.dbench

```
{
  "schema_version": "1.0",
  "criteria": [
    {
      "tguid": "default.dbench.Throughput",
      "reference": {
        "value": 100,
        "operator": "gt"
      }
    },
    {
      "tguid": "default.dbench",
      "min_pass": 1
    }
  ]
}
```

The interpretation of this criteria file is that the measured value of `dbench.Throughput` (the result value) must have a value greater than 100. Also, at least 1 measure under the `default.dbench` test must pass, for the the entire test to pass.

Simple count

```
{
  "schema_version": "1.0",
  "criteria": [
    {
      "tguid": "default",
      "max_fail": 2
    }
  ],
}
```

The interpretation of this criteria file is that the test may fail up to 2 individual test cases, under the `default` test set, and still pass.

Child results

```
{
  "schema_version": "1.0",
  "criteria": [
    {
      "tguid": "syscall",
      "min_pass": 1000,
      "max_fail": 5
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```

    },
    {
      "tguid": "timers",
      "fail_ok_list": ["leapsec_timer"]
    },
    {
      "tguid": "pty",
      "must_pass_list": ["hangup01"]
    }
  ]
}

```

The interpretation of this criteria file is that, within the `syscall` test set, a minimum of 1000 testcases must pass, and no more than 5 fail, in order for that set to pass. Also, in the test set `timers`, if the testcase `leapsec_timer` fails, it will not cause the entire test to fail. However, in the test set `pty`, the testcase `hangup01` must pass for the entire test to pass.

7.34.5 Schema

The schema for the `criteria.json` file is contained in the `fuego-core` repository at: `scripts/parser/fuego-criteria-schema.json`.

Here it is (as of Fuego 1.2):

```

{
  "$schema": "http://json-schema.org/schema#",
  "id": "http://www.fugetest.org/download/fuego_criteria_schema_v1.0.json",
  "title": "criteria",
  "description": "Pass criteria for a test suite",
  "definitions": {
    "criterion": {
      "title": "criterion ",
      "description": "Criterion for deciding if a test (test_set, test_case or ↪measure) passes",
      "type": "object",
      "properties": {
        "tguid": {
          "type": "string",
          "description": "unique identifier of a test (e.g.: Sequential_Output. ↪CPU)"
        },
        "min_pass": {
          "type": "number",
          "description": "Minimum number of tests that must pass"
        },
        "max_fail": {
          "type": "number",
          "description": "Maximum number of tests that can fail"
        },
        "must_pass_list": {
          "type": "array",
          "description": "Detailed list of tests that must pass",

```

(continues on next page)

(continued from previous page)

```

        "items":{
            "type":"string"
        },
        "fail_ok_list":{
            "type":"array",
            "description":"Detailed list of tests that can fail",
            "items":{
                "type":"string"
            }
        },
        "reference":{
            "type":"object",
            "description":"Reference measure that is compared to a result_
↪measure to decide the status",
            "properties":{
                "value":{
                    "type":[
                        "string",
                        "number",
                        "integer"
                    ],
                    "description":"A value (often a threshold) to compare_
↪against. May be two numbers separated by a comma for the 'bt' operator."
                },
                "operator":{
                    "type":"string",
                    "description":"Type of operation to compare against",
                    "enum":[
                        "eq",
                        "ne",
                        "gt",
                        "ge",
                        "lt",
                        "le",
                        "bt"
                    ]
                }
            }
        },
        "required":[
            "value",
            "operator"
        ]
    },
    "required":[
        "tguid"
    ]
},
"properties":{

```

(continues on next page)

(continued from previous page)

```

    "schema_version":{
      "type":"string",
      "description":"The version number of this JSON schema",
      "enum":[
        "1.0"
      ]
    },
    "criteria":{
      "type":"array",
      "description":"A list of criterion items",
      "items":{
        "$ref":"#/definitions/criterion"
      }
    }
  ],
  "required":[
    "schema_version",
    "criteria"
  ]
}

```

7.34.6 Compatibility with previous Fuego versions

The `criteria.json` file replaces the `reference.log` file that was used in versions of Fuego prior to 1.2. If a test is missing a `criteria.json` file, and has a `reference.log` file, then Fuego will read the `reference.log` file and use its data as the the pass criteria for the test.

Previously, Fuego (and it's predecessor JTA) supported pass criteria functionality in two different ways:

- Functional test pass/fail counts
- Benchmark measure evaluations

Functional test pass/fail counts

For functional tests counts of positive and negative results were either hard-coded into the base scripts for the test, as arguments to the `log_compare()` in each test's `test_processing()` function, or they were specified as variables, read from the board file, and applied in the `test_processing()` function.

For example, the `Functional.OpenSSL` test used values of 176 pass and 86 fails (see `fuego-core/tests/Functional.OpenSSL/OpenSSL.sh` in `fuego-1.1`) to evaluate the result of this test.

```

log_compare "$TESTDIR" "176" "${P_CRIT}" "p"
log_compare "$TESTDIR" "86" "${N_CRIT}" "n"

```

But tests in JTA, such as `Functional.LTP.Open_Posix` expected the variables `LTP_OPEN_POSIX_SUBTEST_COUNT_POS` and `LTP_OPEN_POSIX_SUBTEST_COUNT_NEG` to be defined in a the board file for the device under test.

For example, the board file might have lines like the following:

```

LTP_OPEN_POSIX_SUBTEST_COUNT_POS="1232"
LTP_OPEN_POSIX_SUBTEST_COUNT_NEG="158"

```

These were used in the `log_compare` function of the base script of the test like so:

```
log_compare "$TESTDIR" $LTP_OPEN_POSIX_SUBTEST_COUNT_POS "${P_CRIT}" "p"
log_compare "$TESTDIR" $LTP_OPEN_POSIX_SUBTEST_COUNT_NEG "${N_CRIT}" "n"
```

Starting with Fuego version 1.2, these would be replaced with `criteria.json` files like the following:

For `Functional.OpenSSL`:

```
{
  "schema_version": "1.0",
  "criteria": [
    {
      "tguid": 'OpenSSL',
      "min_pass": 176,
      "max_fail": 86
    }
  ]
}
```

For `Functional.LTP.Open_Posix`:

```
{
  "schema_version": "1.0",
  "criteria": [
    {
      "tguid": 'LTP.Open_Posix',
      "min_pass": 1232,
      "max_fail": 158
    }
  ]
}
```

FIXTHIS - should there be 'default' somewhere in the preceding tguids?

Benchmark measure evaluations

For Benchmark programs, the pass criteria consists of one or more measurement thresholds that are compared with the results produced by the Benchmark, along with the operator to be used for the comparison.

In JTA and Fuego 1.1 this data was contained in the `reference.log` file.

7.35 Coding Style

This page described the coding style conventions used in Fuego.

Please adhere to these conventions, so that the code has a more uniform style and it is easier to maintain. Not all code in Fuego adheres to these styles. As we work on code, we will convert it to the preferred style over time. New code should adhere to the preferred style.

Fuego code consists mostly of shell script and python code.

7.35.1 Indentation and line length

We prefer indentation to be 4 spaces, with no tabs.

It is preferred to keep lines within 80 columns. However, this is not strict. If a string constant causes a line to run over 80 columns, that is OK.

Some command sequences passed to the ‘report’ function may be quite long and require that they be expressed on a single line. In that case, you can break them up onto multiple lines using shell continuation lines.

7.35.2 Trailing whitespace

Lines should not end in trailing whitespace. That is: ‘grep ”\$” *’ should always be empty.

You can do this with: ‘grep -R ”\$” *’ in the directory you’re working in, and fix the lines manually.

Or, another method, if you’re using vim, is to add an autocmd to your .vimrc to automatically remove whitespace from lines that you edit.

This line in your ~/.vimrc:

```
autocmd FileType sh,c,python autocmd BufWritePre <buffer> %s/\s\+$/e
```

automatically removes whitespace from all line endings in shell, C, and python files that are saved from vim.

Or, a third method of dealing with this automatically is to have git check for whitespace errors using a configuration option, or a hook. See

<https://stackoverflow.com/questions/591923/make-git-automatically-remove-trailing-whitespace-before-committing#592014>

Also, script files should not end in blank lines.

7.35.3 Shell features

Shell scripts which run on the device-under-test (DUT or board), SHOULD restrict themselves to POSIX shell features only. Do not assume you have any shell features on the target board outside of those supported by ‘busybox ash’.

Try running the program `checkbashisms` on your target-side code, to check for any non-POSIX constructs in the code.

The code in `fuego_test.sh` is guaranteed to run in bash, and may contain bashisms, if needed. If equivalent functionality is available using POSIX features, please use those instead. Please avoid esoteric or little-known bash features. (Or, if you use such features, please comment them.)

Another useful tool for checking your shell code is a program called ‘ShellCheck’. See <https://github.com/koalaman/shellcheck>. Most distributions have a package for `shellcheck`.

There are a few conventions for avoiding using too many external commands in shell scripts that execute on the DUT. To check for a process, use `ps` and `grep`, but to avoid having `grep` find itself, use a wildcard in the search pattern. Like so: ‘`ps | grep [f]oo`’ (rather than ‘`ps | grep foo | grep -v grep`’).

7.35.4 Python style

Python code (such as parser code, the overlay generator, ftc and other helper scripts), should be compliant with <https://www.python.org/dev/peps/pep-0008/>. As with shell code, there is a lot of legacy code in Fuego that is not currently compliant with PEP 8. We will convert legacy code to the correct style as changes are made over time.

Here are a few more conventions for Fuego code:

- Strings consisting of a single character should be declared use single-quotes
- Strings consisting of multiple characters should declared using double-quotes, unless the string contains a double-quote. In that case, single-quotes should be using for quoting, to avoid having to escape the double-quote.

Note that there is a fuego lint test (selftest), called `Functional.fuego_lint`. It only checks a few files at the moment, but the plan is to expand it to check additional code in the future.

7.36 Testplan Reference

In Fuego, a testplan is used to specify a set of tests to execute, and the settings to use for each one.

7.36.1 Filename and location

A testplan is in json format, and can be located in two places:

- As a file located in the directory `fuego-core/overlay/testplans`.
- As a here document embedded in a batch test script (`fuego_test.sh`)

A testplan file name should start with the prefix `“testplan_”`, and end with the extension `“.json”`.

A testplan here document should be preceded by a line starting with `BATCH_TESTPLAN=` and followed by a line starting with `“END_TESTPLAN”`.

7.36.2 Top level attributes

The top level objects that may be defined in a testplan are:

- `testPlanName`
- `tests`
- `default_timeout`
- `default_spec`
- `default_reboot`
- `default_rebuild`
- `default_precleanup`
- `default_postcleanup`

Each of these attributes, except for `‘tests’` has a value that is a string. Here are their meanings and legal values:

The `testPlanName` is the name of this testplan. It must match the filename that holds this testplan (without the `“testplan_”` prefix or `“.json”` extension. This object is required.

`‘tests’` is a list of tests that are part of this testplan. See below for a detailed description of the format of an element in the `‘tests’` list. This object is required.

Default test settings

The testplan may also include a set of default values for test settings. The test settings for which defaults may be specified are:

- timeout
- spec
- reboot
- rebuild
- precleanup
- postcleanup

These values are used if the related setting is not specified in the individual test definition.

For example, the testplan might define a `default_timeout` of “15m” (meaning 15 minutes). The plan could indicate timeouts different from this (say 5 minutes or 30 minutes) for individual tests, but if a test in the testplan doesn’t indicate its own timeout it would default to the one specified as the default at the top level of the testplan.

The ability to specify per-plan and per-test settings makes it easier to manage these settings to fit the needs of your Fuego board or lab.

Note that if neither the individual test nor the testplan provide a default value is not provided, then a Fuego global default value for that setting will be used.

Note that `default_spec` specifies the name of the test spec to use for the test (if one is not specified for the individual test definition). The name should match a spec that is defined for every test listed in the plan. Usually this will be something like “default”, but it could be something that is common for a set of tests, like ‘mmc’ or ‘usb’ for filesystem tests.

See the individual test definitions for descriptions of these different test settings objects.

7.36.3 Individual test definitions

The ‘tests’ object is a list of objects, each of which indicates a test that is part of the plan. The objects included in each list element are:

- testName
- spec
- timeout
- reboot
- rebuild
- precleanup
- postcleanup

All object values are strings.

TestName

The ‘testName’ object has the name of a Fuego test included in this plan. It should be the fully-qualified name of the test (that is, it should include the “Benchmark.” or “Functional.” prefix.) This attribute of the test element is required.

Spec

The ‘spec’ object has the name of the spec to use for this test. It should match the name of a valid test spec for this test. If ‘spec’ is not specified, then the value of “default_spec” for this testplan will be used.

Timeout

The timeout object has a string indicating the timeout to use for a test. The string is positive integer followed by a single-character units-suffix. The units suffixes available are:

- ‘s’ for seconds
- ‘m’ for minutes
- ‘h’ for hours
- ‘d’ for days

Most commonly, a number of minutes is specified, like so:

- “default_timeout” : “15m”,

If no ‘timeout’ is specified, then the value of ‘default_timeout’ for this testplan is used.

Reboot

The ‘reboot’ object has a string indicating whether to reboot the board prior to the test. It should have a string value of ‘true’ or ‘false’.

Rebuild

The ‘rebuild’ object has a string indicating whether to rebuild the test software, prior to executing the test. The object value must be a string of either ‘true’ or ‘false’.

If the value is ‘false’, then Fuego will do the following, when executing the test:

- If the test program is not built, then build it
- If the test program is already built, then use the existing test program

If the value is ‘true’, then Fuego will do the following:

- Remove any existing program build directory and assets
- Build the program (including fetching the source, unpacking it, and executing the instructions in the test’s “test_build” function)

Precleanup

The ‘precleanup’ flag indicates whether to remove all previous test materials on the target board, prior to deploying and executing the test. The object value must be a string of either ‘true’ or ‘false’.

Postcleanup

The ‘postcleanup’ flag indicates whether to remove all test materials on the target board, after the test is executed. The flag value must be a string of either ‘true’ or ‘false’.

7.36.4 Test setting precedence

Note that the test settings are used by the plan at job creation time, to set the command line arguments that will be passed to `ftc run-test` by the Jenkins job, when it is eventually run.

A user can always edit a Jenkins job (for a Fuego test), to override the test settings for that job.

The precedence of the settings encoded into the job definition at job creation time are:

- Testplan individual test setting (highest priority)
- Testplan default setting
- Fuego default setting

The precedence of settings at job execution time are:

- ‘ftc run-test’ command line option setting (highest priority)
- Fuego default setting

7.37 Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)

7.37.1 Fuego Test System

Welcome to Fuego!

Fuego is a test system specifically designed for embedded Linux testing. It supports automated testing of embedded targets from a host system, as its primary method of test execution.

Fuego consists of a host/target script engine, and over 100 pre-packages tests. These are installed in a docker container along with a Jenkins web interface and job control system, ready for out-of-the-box Continuous Integration testing of your embedded Linux project., and over 100 pre-packaged

The idea is that in the simplest case, you just add your board, select or install a toolchain, and go!

Introduction presentation

Tim Bird gave some talks introducing Fuego, at various conferences in 2016. The slides and a video are provided below, if you want to see an overview and introduction to Fuego.

The slides are here: [Introduction-to-Fuego-LCJ-2016.pdf](#), along with a [YouTube video](#). You can find more presentations about Fuego on our wiki at: <http://fuegotest.org/wiki/Presentations>.

Getting Started

There are a few different ways to get started with Fuego:

1. Use the *Fuego Quickstart Guide* to get Fuego up and running quickly.
2. Or go through our *Install and First Test* tutorial to install Fuego and run a test on a single “fake” board. This will give you an idea of basic Fuego operations, without having to configure Fuego for your own board.
3. Work through the documentation for *Installation*

Where to download

Code for the test framework is available in 2 git repositories:

- <https://bitbucket.org/fuegotest/fuego/>
- <https://bitbucket.org/fuegotest/fuego-core/>

The fuego-core directory resides inside the fuego directory. But normally you do not clone that repository directly. It is cloned for you during the Fuego install process. See the *Fuego Quickstart Guide* or the *Installing Fuego* page for more information.

Documentation

See the index below for links to the major sections of the documentation for Fuego. The major sections are:

- *Tutorials*
- *Installation and Administration*
- *User Guides*
- *Developer Resources*
- *API Reference*

Resources

Mailing list

Fuego discussions are held on the fuego mailing list:

- <https://lists.linuxfoundation.org/mailman/listinfo/fuego>

Note that this is a new list (as of September 2016). Previously, discussions about Fuego (and its predecessor JTA) were held on the ltsi-dev mailing list:

- <https://lists.linuxfoundation.org/mailman/listinfo/ltsi-dev>

Presentations

A number of presentations have been given on the Fuego test framework, and related projects (such as its predecessor JTA, and a derivative project JTA-AGL).

See the [Presentations](#) page on the Fuego wiki for a list of presentations that you can read or view for more information about Fuego.

Vision

The purpose of Fuego is to bring the benefits of open source to the testing process.

It can be summed up like this:

Note: Do for testing what open source has done for coding

There are numerous aspects of testing that are still done in an ad-hoc and company-specific way. Although there are open source test frameworks (such as Jenkins or LAVA), and open source test programs (such as cyllictest, LTP, linuxbench, etc.), there are lots of aspects of Linux testing that are not shared.

The purpose of Fuego is to provide a test framework for testing embedded Linux, that is distributed and allows individuals and organizations to easily run their own tests, and at the same time allows people to share their tests and test results with each other.

Historically, test frameworks for embedded Linux have been difficult to set up, and difficult to extend. Many Linux test systems are not easily applied in cross or embedded environments. Some very full frameworks are either not viewed as processor-neutral, and are difficult to set up, or are targeted at running tests on a dedicated group of boards or devices.

The vision of open source in general is one of sharing source code and capabilities, to expand the benefits to all participants in the ecosystem. The best way to achieve this is to have mechanisms to easily use the system, and easily share enhancements to the system, so that all participants can use and build on each others efforts.

The goal of Fuego is to provide a framework that any group can install and use themselves, while supporting important features like cross-compilation, host/target test execution, and easy test administration. Test administration consists of starting tests (both manually and automatically), viewing test results, and detecting regressions. Ease of use is critical, to allow testers to use tests that are otherwise difficult to individually set up, configure, and interpret the results from. It is also important to make it very easy to share tests (scripts, configuration, results parsing, and regression detection methods).

Some secondary goals of this project are the ability for 3rd parties to initiate or schedule tests on our hardware, and the ability to share our test results with others.

The use of Jenkins as the core of the test framework already supports many of the primary and secondary goals. The purpose of this project is to augment the Jenkins system to support embedded configurations of Linux, and to provide a place for centralized sharing of test configurations and collateral.

There is no such thing as a “Linux Test distribution”. Fuego aims to be this. It intends to provide test programs, a system to build, deploy and run them, and tools to analyze, track, and visualize test results.

For more details about a high-level vision of open source testing, please see [OSS Test Vision](#).

Other Resources

Historical information

<http://elinux.org/Fuego> has some historical information about Fuego.

Related systems

See Other test systems for notes about other test frameworks and comparisons between Fuego and those other systems.

Things to do

Looking for something to do on Fuego? See the Fuego wiki for a list of projects, at: [Fuego To Do List](#)

7.37.2 Sandbox

This page is for testing different elements of reStructuredText markup, in the Fuego documentation. This is intended to be used during the conversion from wiki pages, to make sure all important formatting is preserved.

7.37.3 Page Level Header (H1)

Here is some text

Chapter Level Header (H2)

This is the start of a level 2 section

Level 3 header

Does this actually do a level 3 header?

Level 4 header

Some content here at level 4

Level 5 header

Some content here at level 5

Some Example Markup

Here is a **bold** word, and an *italic* word. Also here is a `filename.txt`, and a command `-a arg`.

Here is a glossary terms: board

Admonition test

Reference test

test 1

Sandbox2_

Here is an attempt to refer to a page: ``Sandbox2`_`

Result:

- cover text of ``Sandbox2`_`
- URL of `.../_build/html/Sandbox.html#id1` - FAIL

cover text of

test 2

Pointer to slandbox page1 `<sand1>_`

Here is another attempt to refer to a page. Try to refer to a section on a page, with cover text: `Pointer to slandbox page1`

Result:

- cover text of `Pointer to slandbox page1`
- URL of `.../_build/html/sand1` - FAIL

test 3

Reference: `:ref:`Pointer to slandbox page3 <sndbx2>``

Here is another way to refer to a section on a page, with cover text: `Pointer to slandbox page3`

Result from default Ubuntu 16.03 Sphinx:

- cover text of “Pointer to slandbox page3”
- NO URL! - FAIL

Result from python3 venv py3-sphinx Sphinx installation:

- cover text of “Pointer to slandbox page3”
- URL of `.../_build/html/Sandbox2.html#sndbx2` - PASS

test 4

Reference: `:ref:`Pointer to slandbox2 page4 <Sandbox2>``

Here is another way to refer to a section on a page, with cover text: `Pointer to slandbox2 page4`

Result:

- cover text of `Pointer to slandbox2 page4`
- NO URL! - FAIL

test 5

Reference ``Pointer to slandbox2 page5 <Sandbox2.html>`_`

Here is another way to refer to a section on a page, with cover text: `Pointer to slandbox2 page5`

Result:

- cover text of `Pointer to slandbox page5`
- URL = `Sandbox2.html` - PASS

Conclusion. There doesn't seem to be a way to refer to a page or a section heading on a page, unless it is marked with an anchor. (unless you reference the page with it's .html extension)

test 6

Reference ``Pointer to slandbox2 test6 <chapheader2>``

Here is another way to refer to a section on a page, with cover text: `Pointer to slandbox2 test6`

Result:

- cover text of `Pointer to slandbox test6`
- URL = `.../_build/html/Sandbox2.html#chapheader2` - PASS

Which ways worked?

- Test 3 - worked it should have
- Test 5 - but it's gross
- Test 6 - is preferred.

Toctree test

toctrees apparently refer to file (page) names. The items put into the tree are the section headings from those pages

Here's another toctree - this time with a caption

Following this is a hidden toctree

I can keep doing this all day!!

Notice there is no anchor or label before this section heading

These are items from the `Sandbox2` page.

7.37.4 Sndbx2

This page header doesn't have the same name as the page.

This page is for testing different elements of reStructuredText markup, in the Fuego documentation. This is intended to be used during the conversion from wiki pages, to make sure all important formatting is preserved.

7.37.5 Page Level Header2

Here is some text

Chapter Level Header2

Reference test

Test 1

Reference: `Sandbox`_

Here is how you refer to a page: `**Sandbox`_**

Result: .../_build/html/Sandbox2.html#id4 - FAIL

Test 2

Reference: `Sandbox2`_

Here is another way to refer to a page: `**Sandbox2`_**

Result: .../_build/html/Sandbox2.html#id6 - FAIL

Test 3

Reference: Sandbox2_

Here is another way to refer to a page: **Sandbox2`_**

Result:

- cover = ???
- URL = .../_build/html/Sandbox2.html#id8 - FAIL

Test 4

Description: anchor reference using underscore

Reference: `Pointer to slandbox page1 <sand1>`_

Here is one way to refer to a section on a page, with cover text: [Pointer to slandbox page1](#)

Result:

- cover - Pointer to slandbox page1
- URL - .../_build/html/sand1 - FAIL

Test 4

Here is another way to refer to a section on a page, with cover text:

This uses a ref:

Reference: :ref:`Pointer to slandbox page2 <sand1>`

Here is the ref: *Pointer to slandbox page2*

Result:

- cover = Pointer to slandbox page2
- URL = .../_build/html/Sandbox.html#sand1 - PASS

Conclusion:

- you MUST have an anchor to use a ref
- only a :ref: gives you the page name (Sandbox.html) and the anchor ref (sand1)

Test 3

Here is one way to refer to a section on a page, with cover text: This one uses trailing _ and the page name: *Pointer to slandbox2 page*

Here is one way to refer to a section on a page, with cover text: This one uses an href and the page name: *Pointer to slandbox2 page*

Here is another way to refer to a section on a page, with cover text: This one uses a trailing _ and the section name: *Pointer to slandbox2 page header*

Here is another way to refer to a section on a page, with cover text: This one uses a ref and the section name: *Pointer to slandbox2 page header*

Which ways worked?

Toctree test

toctrees apparently refer to file (page) names. The items put into the tree are the section headings from those pages

Here's another toctree - this time with a caption

Following this is a hidden toctree

I can keep doing this all day!!