JTA Quickstart guide

Cogent Embedded Inc.
http://cogentembedded.com

June 4, 2014

Abstract

This document is the complete reference for JTA test system. It contains installation guide, basic usage guide, step-by-step guides for adding test board and integrating a test. In later chapters it is described how test overlays and pdf test report generation work.

Contents

1 Introduction

Testing evaluation boards is not easy. There are a number of software products made for that very purpose, the most eminent of which is Lava.

On the other hand, this framework was designed to provide a core meeting a few points:

- It is usable out of the box: 60+ prepackaged tests, benchmark statistics, plotting and reports generation.
- It is highly custmizable from front-end side (thanks to Jenkins that have tons of plugins) and from backend side, that relies on simple core written in bash, so that allows it to be easily customized;
- It allows for flexible test configuration using such notions as *test specification* and *test plan* 7;
- It allows to run a batch of tests and generate report 8);
- It does not impose any demands on boards on distributions;
- It allows easy, yet flexible board setup. All you need is just to define some environment variables (block devices/mount points, IP addr etc) in board config file.

As you can see, our goal is to provide flexible framework with seemless customization and out of box experience.

2 Installation

2.1 Prerequisites

- Debian **Wheezy** for **AMD64** architecture.
  It can be fetched from here;
- Web browser with **javascript** and **CSS** support.

2.2 Running install script

- Get JTA release via git:
  `git clone https://cogentembedded@bitbucket.org/cogentembedded/jta-public.git`
- Run install script under root or sudo: `./debian_install`[1]

When script finishes, JTA web interface will be available on local machine at port 8080.

2.3 Installing toolchains and sysroots

You need toolchains and sysroots to build tests for different platforms.

2.3.1 Using meta-jta OE layer for generating toolchain

We ship toolchains for koelsch and minnow boards that we build from poky from our custom layer. If you already use OpenEmbedded from building rootfs for your system yoy can use our layer to generate toolchain and sysroot (using `bitbake meta-toolchain`) with all libraries and headers needed from building tests. See Poky Documentation for more information.

Toolchains and sysroots are usually stored in `/home/jenkins/tools` folder.

2.4 Configuring tools.sh file

`/home/jenkins/scripts/tools.sh` file is used to setup paths and compile flags for each platform.

For poky-generated toolchains one should source envirnoment file and set the following variables:

- `SDKROOT` - path to rootfs
- `PREFIX` - gcc prefix, like `arm-blabla-linux-gnueabi`
- `HOST` - like `PREFIX`

Also not code saving original `$PATH` to `$ORIG_PATH` since envirnoment script changes it.

See [L. 3] for example.

[1] During the proecess script will call dpkg-reconfigure dash, asking a question about using dash. You must answer "no" to the dialogue.

2.4.1 Using custom toolchain

For using custom toolchain you *additionally* must define the following variables: `PATH`, `PKG_CONFIG_SYSROOT_DIR`, `PKG_CONFIG_PATH`, `CC`, `CXX`, `CPP`, `AS`, `LD`, `RANLIB`, `AR`, `NM`, `CFLAGS`, `CXXFLAGS`, `LDFLAGS`, `CPPFLAGS`, `ARCH`, `CROSS_COMPILE`.

You can use `environment-setup-core2-32-osv-linux` script as reference.

3 Boards configuration

In this document we will use such notions as *targets* and *boards*. Here is what they mean:

Target or Node is understood as a front-end entity.

Board is denoted as a back-end entity.

Board configuration is done in `/home/jenkins/overlays/boards/<boardname>.board`, where `<boardname>` is the respective name of the target.

3.0.2 Adding a target in frontend

The simpliest way to add a Node is to copy it from existing one. We provide *template-dev* board for that purpose.

1. Click Target status
2. Click New node
3. Fill in the *Node name* input field.
4. Choose *Copy Existing Node*. And enter name of source node, namely, *Generic*
5. You will be forwarded to the new node configuration page. Locate *Environment variables* section in *Node Properties*. There you should path to board config file [3.0.3] to the variable **BOARD_OVERLAY**.

3.0.3 Board config file

A board file has simple format close to the bash shell. It has two extra syntactic constructs:

`inherit` is used to read and inherit the base[2] class config file.
Example: `inherit "base-file"`.

`override` is used for overriding base variables and functions.
Example: `override ROOTFS_LOGREAD "cat_messages"`.

The following is the step-by-step description of all mandatory environment variables should be set:

`TRANSPORT`: defines how JTA should communicate with the board. Currently only `ssh` is supported;

`IPADDR`: IP address for board;

`LOGIN`: user name for ssh login;

[2] See 3.0.4 for more info.

`PASSWORD`: password for ssh login;

`JTA_HOME`: path to the directory to which the tests will be copied before running;

`PLATFORM`: architecture of the board. Currently `ia32`, `arm` and `mips` are supported.

The following variables specify devices and mount points that are used by some filesystem tests: `SATA_DEV`, `SATA_MP`, `USB_DEV`, `USB_MP`, `MMC_DEV`, `MMC_MP`.

3.0.4 Base class

It is a special shell script, containing definition of basic parameters. You can have as many base classes as you want.

base-distrib is the default class which should inherited from in boards config. It is located in `overlays/base/base-distrib.jtaclass` file.

It sets the following variables to "`default`" value:

`ROOTFS_FWVER` returns kernel version

`ROOTFS_REBOOT` is reboot command

`ROOTFS_KILL` is pkill command

`ROOTFS_DROP_CACHES` is vm caches drop command

`ROOTFS_SYNC` is sync command

`ROOTFS_LOGGER` is logger command

`ROOTFS_STATE` returns system state (uptime, free, df, mount, ps, /proc/interrupts)

`ROOTFS_LOGREAD` returns system logs

`ROOTFS_OOM` adjusts OOM killer

If you want to change the commands you can add the cases handling to `functions.sh` script, at places where these commands are used.

`OF.NAME OF.DESCRIPTION` is the service fields that set base class name and description accordingly.

4 Running tests

4.1 Running single tests

1. From the main page open *Functional* or *Benchmarks* tab.
2. Click on the test name.
3. Click on *Run test now*.

Here you can set test run parameters. The most relevant are:

Device: Choose a target the test will be run on.

Reboot: If checked target device will be rebooted before running test.

Rebuild: Rebuild[3] the test.

TESTPLAN: (optional) Derive test parameters from test specifications from chose testplan. For testplans see [7]

Press *Run test* button. The test is scheduled for running. If no tests are executed on target it will be run immediately. It will appear in the left frame name *Test run history*. There you can see all this specific test results (including currenlty running one). They can be of a few types:

Solid green circle: The test has been successefully run.

Solid red circle: The test has failed.

Flashing circle: The test is currently running.

If you point mouse over the date of test run the pop-up menu appears from where you can go to *Console output*. There you can view the complete log of test run.

4.2 Running a group of tests

1. From the main page open *Batch runs*.
2. Click on *Run SELECTED tests on SELECTED targets*.
3. Click on *Run test now*.
4. Choose a target[4]
5. Mark tests/benchmarks you would like to run.

[3] Some tests require rebuilding if their parameters were changed.

[4] Reports generation is implemented only for the first target selected and only for Run SELECTED tests on SELECTED batch run

6. Enter test plan to `TESTPLAN` variable. Test plan is mandatory for batch runs.

7. Click on *Run test* button.

The batch test run is scheduled for running on target. On the test page there is a list of running tests (their statuses are the same as in [4.1]).

When batch run is finished you can view generated PDF report.

4.3 Viewing PDF reports

1. From the batch test run page click on *Workspace link*

2. Click on pdf_reports folder.

3. Click the bottommost pdf file.
   It has `<target>.<date>.<testrun>.json.xml.pdf` format.

5 Adding a sample test

This section describes how to integrate tests to OSV. We will add a simple test that calls bc computing a value passed through spec parameter.

5.1 Adding Test Plan files

Create `/home/jenkins/overlays/testplans/testplan_bc_exp1.json`[L. 1] and `testplan_bc_exp2.json`[L. 2] files.

As you can see we've created two testplan files which reference two specs. Testplan can reference multiple specs for different tests, so for example we could run all filesystem tests with specific block device.

5.2 Adding spec file

Create `/home/jenkins/overlays/test_specs/Benchmark.bc.spec`[L. 4] file.

This spec file contatins two cases: `bc-exp1` that generates `EXPR1`, `EXPR2` variable and assignes it `"2*2"`, `"3*3"` values [5] and `bc-spec-exp2` that does the same but with `"2+2"` and `"3+3"` values. These variables is inteded to be used inside test script for controlling different test cases. And we will use it as a parameter to `bc-device.sh` script.

You don't usually need more than one spec files, because all different cases can be listed in one file.

5.3 Adding test script

Test script is the bash file that runs when test is executed on target. Create it[L. 5] with the path `/home/jenkins/tests/Benchmark.bc/bc-script.sh`. This file should meet a strict format with following definitions:

`tarball` name of the tarball;

`test_build` should contain test build commands;

`test_deploy` should contain commands that deploy test to device;
put command is usually used;

`test_run` should contain all steps for actual test execution.

Generic benchmark/test script can be sourced if test meets common patterns.
In this particular example `benchmark.sh` is sourced that will execute these steps (and some other like overlay prolog file and reports generation).

For testing purposes we will use a simple script that is executed on device. It accepts two parameters, calls *bc* with them and produces an output. Create `bc-script.tar.gz` tarball containing a folder with `bc-device.sh`[L. 6] file.

[5] Any variable defined in board config file[3.0.4] or in (inherited) base file[3.0.3] can be used. For example `$MINNOW_SATA_DEV`

When benchmark is finished results parsing phase is started. **Each** benchmark (not Functional test) should provide a special python parsing script called `parser.py` that defines how to parse results. All you should do is to fill a `cur_dict` dictionary with `{subtest: value}` pairs and call `plib.process_data` with respecitve arguments:

`ref_section_pat` : regexp that describes the format of threshold expressions

`cur_dict:` : dictionary containing `{subtest: value}` pairs with test results;

`m:` plot type. 's' - single, 'm' - multiple

`label:` axis label

See [L. 7] for a simple script that parses two *bc* outputs.

Core script `common.py` checks the values to agree with *reference values* that should be in `reference.log` file in the directory where main test script resides. See [L. 8] for sample `reference.log` file asserts both results must be greater that 0.

Test integration is complete. Now you should be able to locate test under *Benchmarks* tab in main page.

5.4 Adding test to frontend

The simpliest way to add a benchmark in frontend is using one as a template.

1. From main page click on *New Test*;
2. Fill in *Test name* input field;
3. Choose *Copy existing Test* combo box;
4. Enter test name to the *Copy from* text field. For example, *Benchmark.bonnie*;
5. Press *OK* button.

You will be forwarded to the test configuration page. There are a lot of parameters there, but you only need to set up a few of them:

Description: Textual description of the test;

TESTPLAN: (a string parameter) path to the test plan. Not mandatory. But we will use one for that sake of demonstration. Put `testplans/testplan_bc_exp1.json` there.

Execute shell: bash script that will be executed when test is run.
Put `source ../tests/$JOB_NAME/bc-script.sh` there.

5.4.1 Adding plot paramters to `tests.info`

Plot plugin needs to know which parameters it should display. It uses `tests.info` file for that purpose. Open `/home/jenkins/logs/tests.info` and add the following line: `"bc":["result1","result2"]`. Make sure you meet json syntax this file uses.

This line says to draw *result1* and *result2* values on the plot.

5.5 Conclusion

So, below is the list of all components our benchmark uses.

spec file `Benchmark.bc.spec`[L. 4] that contain list of various options that generate variables for testing;

testplan files `testplan_bc_exp{1,2}.json` [L. 1], [L. 2] that contain lists of specifications should be used for test(s);

test script `bc-script.sh`[L. 5] that runs all top-level commands;

tarball with `bc-device.sh`[L. 6] file that does actual testing on device;

parser.py [L. 7] that parses the results and gives them to core parsing component that prepares data for plots and reports;

reference.log [L. 8] that contains reference values then benchmark results are checked against;

tests.info should be modified to include values should be drawn on plot.

6 Overlays

This section describes how overlays are implemented.

7 Test plans

This section describes how test plans are implemented.

8 Reports

This section describes how reports are implemented.

9 Listings

```
{
    "testPlanName": "testplan_bc",
    "tests": [
        {
            "testName": "Benchmark.bc",
            "spec": "bc-exp1"
        }    ]
}
```

Listing 1: `testplan_bc_exp1.json` file

```
{
    "testPlanName": "testplan_bc",
    "tests": [
        {
            "testName": "Benchmark.bc",
            "spec": "bc-exp2"
        }    ]
}
```

Listing 2: `testplan_bc_exp2.json` file

```
if [ "${PLATFORM}" = "intel-minnow" ];
then
        SDKROOT=/home/jenkins/tools/intel-minnow/
           sysroots/core2-32-osv-linux/
        # environment script changes PATH in the way
           that python uses libs from sysroot which is
            not what we want, so save it and use later
        ORIG_PATH=$PATH
        PREFIX=i586-osv-linux
        source /home/jenkins/tools/intel-minnow/
           environment-setup-core2-32-osv-linux

        HOST=arm-osv-linux-gnueabi

        unset PYTHONHOME
        env -u PYTHONHOME
```

Listing 3: `intel minnow tools` section

```
  {
    "testName": "Benchmark.bc",
    "specs":
```

```
    [
        {
            "name":"bc-exp1",
            "EXPR1":"2*2",
            "EXPR2":"3*3"
        },
        {
            "name":"bc-exp2",
            "EXPR":"2+2",
            "EXPR2":"3+3"
        }
    ]
}
```

Listing 4: `Benchmark.bc.spec` file

```
#!/bin/bash

tarball=bc-script.tar.gz

function test_build {
    echo "test compiling (should be here)"
}

function test_deploy {
   put bc-device.sh  $JTA_HOME/jta.$TESTDIR/
}

function test_run {
    assert_define BENCHMARK_BC_EXPR1
    assert_define BENCHMARK_BC_EXPR2
    report "cd $JTA_HOME/jta.$TESTDIR; ./bc-device.sh
       $BENCHMARK_BC_EXPR1 $BENCHMARK_BC_EXPR1"
}
. ../scripts/benchmark.sh
```

Listing 5: `bc-script.sh` file

```
#!/bin/bash

BC_EXPR1=$1
BC_EXPR2=$1

BC1=`echo $BC_EXPR1 | bc`
BC2=`echo $BC_EXPR2 | bc`
echo "$BC1,$BC2"
```

Listing 6: `bc-device.sh` file

```
#!/bin/python

import os, re, sys, json

sys.path.insert(0, '/home/jenkins/scripts/parser')
import common as plib

cur_dict = {}
cur_file = open(plib.CUR_LOG,'r')
print "Reading current values from " + plib.CUR_LOG +
    "\n"

ref_section_pat = "^\[[\w_ .]+.[gle]{2}\]"

raw_values = cur_file.readlines()
results = raw_values[-1].rstrip("\n").split(",")
cur_file.close()

cur_dict["result1"] = results[0]
cur_dict["result2"] = results[1]

sys.exit(plib.process_data(ref_section_pat, cur_dict,
    's', 'value'))
```

Listing 7: `parser.py` file

```
[result1|ge]
0
[result2|ge]
0
```

Listing 8: `reference.log` file