Board Farm APIs for Automated Testing of Embedded Linux

Harish Bansal
Technical Engineer
Timesys

Tim Bird
Principal Software Engineer
Sony Electronics

Abstract

For years, designers of automated testing systems have used ad-hoc designs for the interfaces between a test, the test framework and board farm software, and the device under test. This has resulted in a situation where hardware tests cannot be reused from one lab to another.

This talk presents a proposal for a standard API between automated tests and board farm management software. The idea is to allow a test to query the farm about available bus connections, attached hardware and monitors, and other test installation infrastructure. The test can then allocate and use that hardware, in a lab-independent fashion. The proposal calls for a dual REST/command-line API, with support for discovery, control and operation - of hardware and network resources. It is hoped that establishing a standard in this area will allow for the creation of an ecosystem of shareable hardware tests and board farm software.

Agenda

- Problem statement
- Introduction to Embedded Board Farm Cloud
- Use cases for REST API
- Board FARM API details
- Issues encountered
- What’s next?

Problem Statement

- There are many tests but no standardized way of running tests on physical devices
- There are many different Test Frameworks
- There are a few Board Farm frameworks but there is no standardized way to use different Test Frameworks or run tests
- Every farm implements test infrastructure differently
  - Many labs use ad-hoc infrastructure
  - Tests written for one lab do not work in another lab
- Nobody can share tests

Solution:

- Creating a standard method to access a Board Farm allows:
  - Board farms infrastructure technologies can evolve separately from the interface to the farm
  - Tests can be written that work in more than one lab
  - Test Frameworks can work with more than one lab

Examples of Hardware/Software integration tests

- GPIO test, Serial Port test
  - Need to control two endpoints
    - One on device under test (DUT) and one external endpoint
- Audio playback test
  - Need to control two endpoints
    - One on device under test (DUT) and a capture device
- Power measurement (via external power monitor)
  - Need to control two endpoints:
    - Application or workload profile on DUT
    - Capture of power measurement data on external power monitor
- USB connect/disconnect (robustness) testing
  - Need to control two endpoints:
    - Application or monitor on DUT
    - USB hardware external to board (drop/re-connect vbus)

Examples of multi-device orchestration

- Netperf test
  - Need to manage two endpoints
    - Netperf client (on DUT)
    - Netperf server (off DUT)
      - May want exclusive use of netperf server for a single node or set of nodes during a test
      - Need to discover server address (specific to lab)
- Boot test
  - Need to manage many devices (DUT, storage, serial, and power controller)
    - DUT: Provision the DUT
    - Storage controller: install kernel and/or root fs
    - Serial controller: capture DUT serial line (for console output)
    - Power controller: turn power off/on

High Level Concept 1 – API between framework and lab

Test Framework A
CLI

Test Framework B
APIs

Test Framework C
APIs

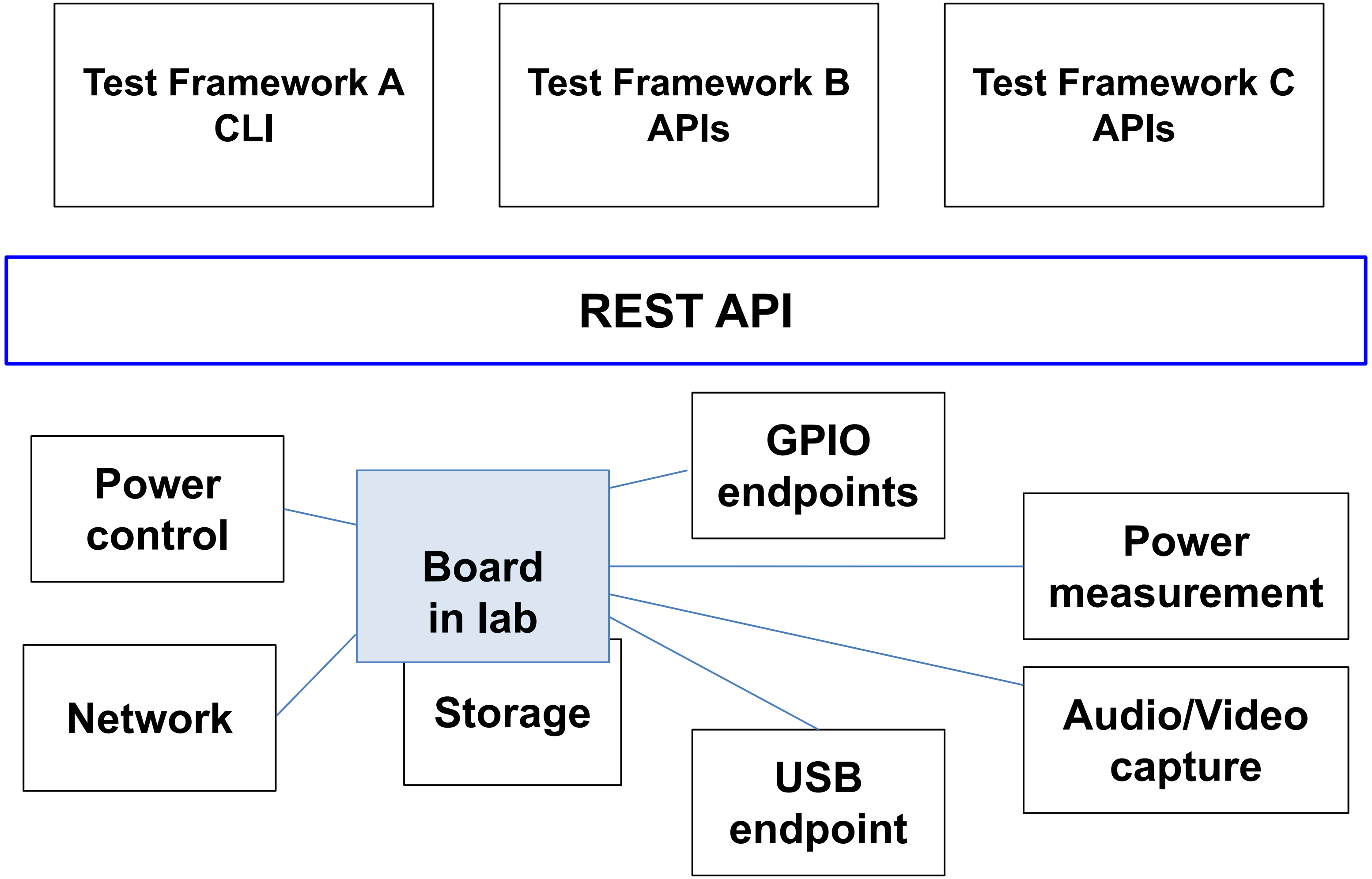
REST API

Board Farm 1

Board Farm 2

Board Farm 3

High Level Concept 2 – API between test and lab

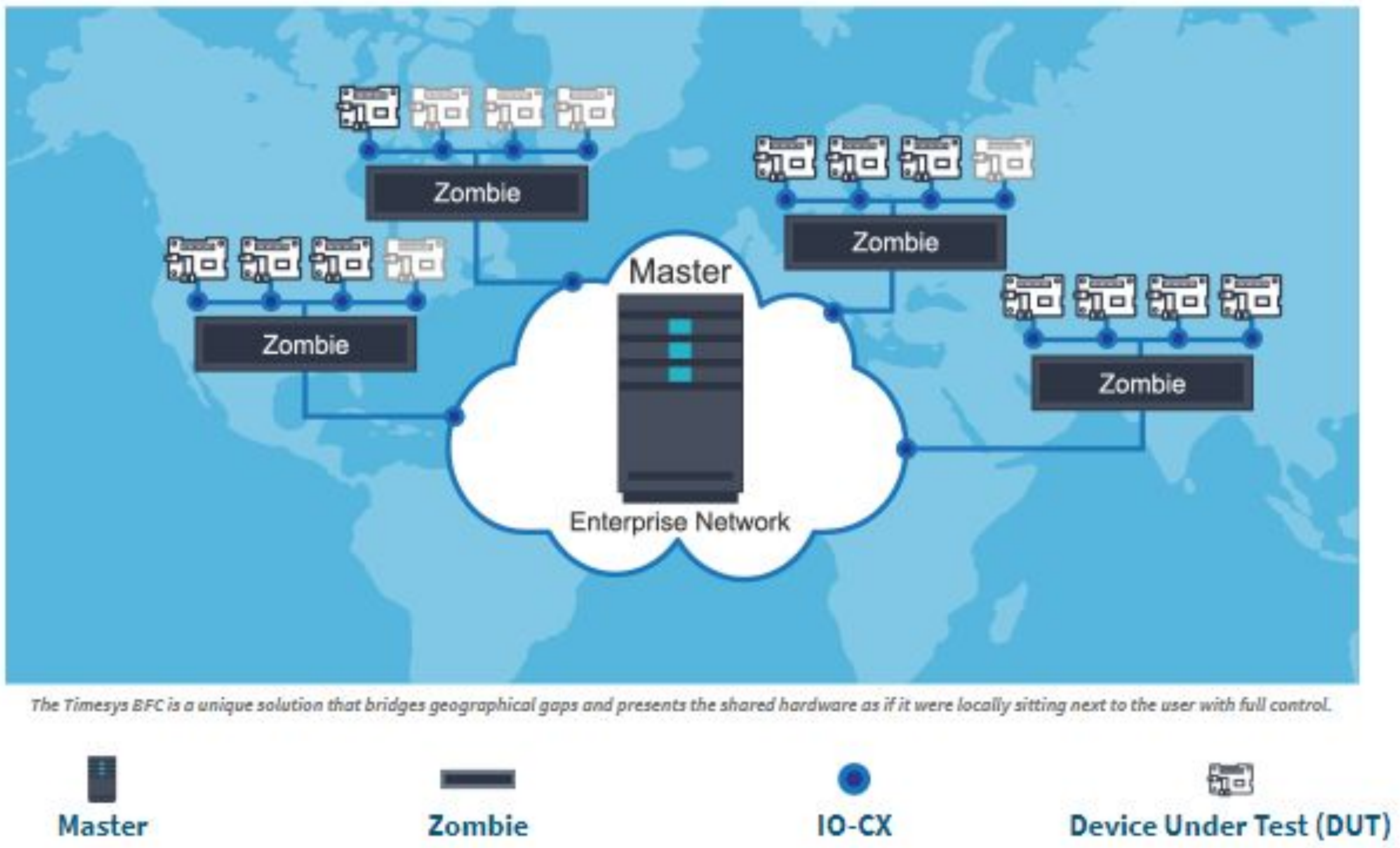


Fuego/EBFC REST-API elements

- API proposal
  - 3 parts
    - REST API
    - Command line interface
    - Environment variables
- REST API based on https and JSON
  - Extension to LAVA REST API
  - Only requires curl and jq
- Command line tool
  - Same operations as REST API
  - Suitable for automated use, as well as human interactive use
- Environment variables
  - Used to communicate values to test program on target
  - Stored in /etc/test-config, or passed in program environment

Introduction to Timesys' Embedded Board Farm Cloud

Timesys Embedded Board Farm Cloud Architecture

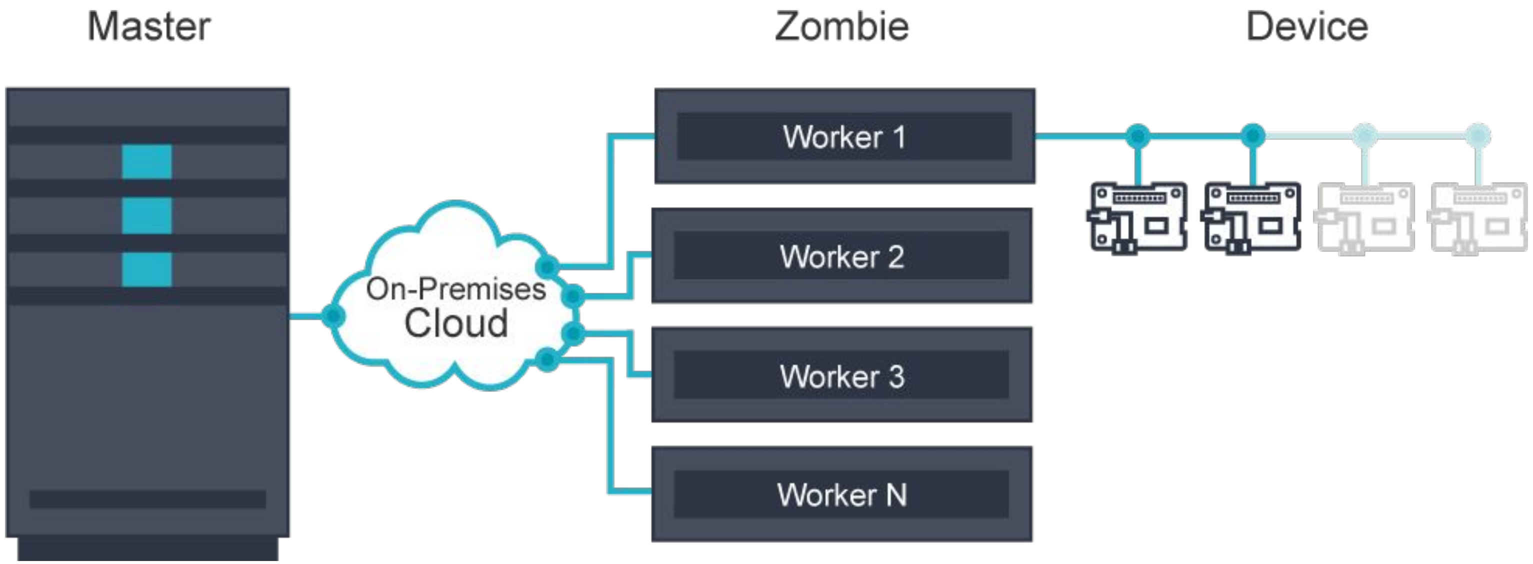


Components

- EBF Master
- Zombie (Lab Controller)
- IO-CX (Lab Controller)
- **Your Boards (DUT)**

EBF Master

- Multi-user Access from anywhere
- Board Dashboard
- Image and File transfer management
- Centralized Board Management
- User management
- Zombie management
- Standard off-the-shelf PC/Server
- EBF Master Docker



Lab Controllers

Zombie

- Zombie (red)
- App/Test Server (blue)

IO-CX

- Power Control
- USB hotplugs
- Ethernet hotplugs
- SDMUX
- USB MUX
- I2C bus connector
- GPIO connector

Board/Device Management

- Sign in
- Visit All Devices
- Allocate the device
- Launch Console
- Retire Device
- Visit My Devices

Active Devices - excluding retired

All Devices | Active Devices | My Devices

Actions

Launch Side By Side Console | Refresh IOCX Status

Show 50 entries — Search

Device Name	Device type	Device status	Assigned To	tags	Zombie Worker	Device Port	IOCX Connected	Mux Mode	Actions
am3354_respironics-1	am335x	Idle	—		BFZombie1	6	False	NA	
am335x_evm-1	am335x	Idle	bill.canfield		BFZombie7	9	False	NA	
am3517_evm-1	am35XX	Idle	—		BFZombie1	3	False	NA	
am437x_evm-1	am437x	Idle	zigiria.kalmara		BFZombie2	4	False	NA	
amcc405gp-1	PowerPC	Idle	—		BFZombie2	8	False	NA	
amcc460ex-1	PowerPC	Idle	jayachandra.k		BFZombie2	12	False	NA	
android_raspi3-1	RasPi3	Idle	shyam.prasath		BFZombie7	3	True	SDMUX	
ARK_1124H-1	intel_atom_e3940_q_SoC	Idle	brian.bender		BFZombie2	6	False	NA	

My Devices

All Devices | Active Devices | My Devices

Actions

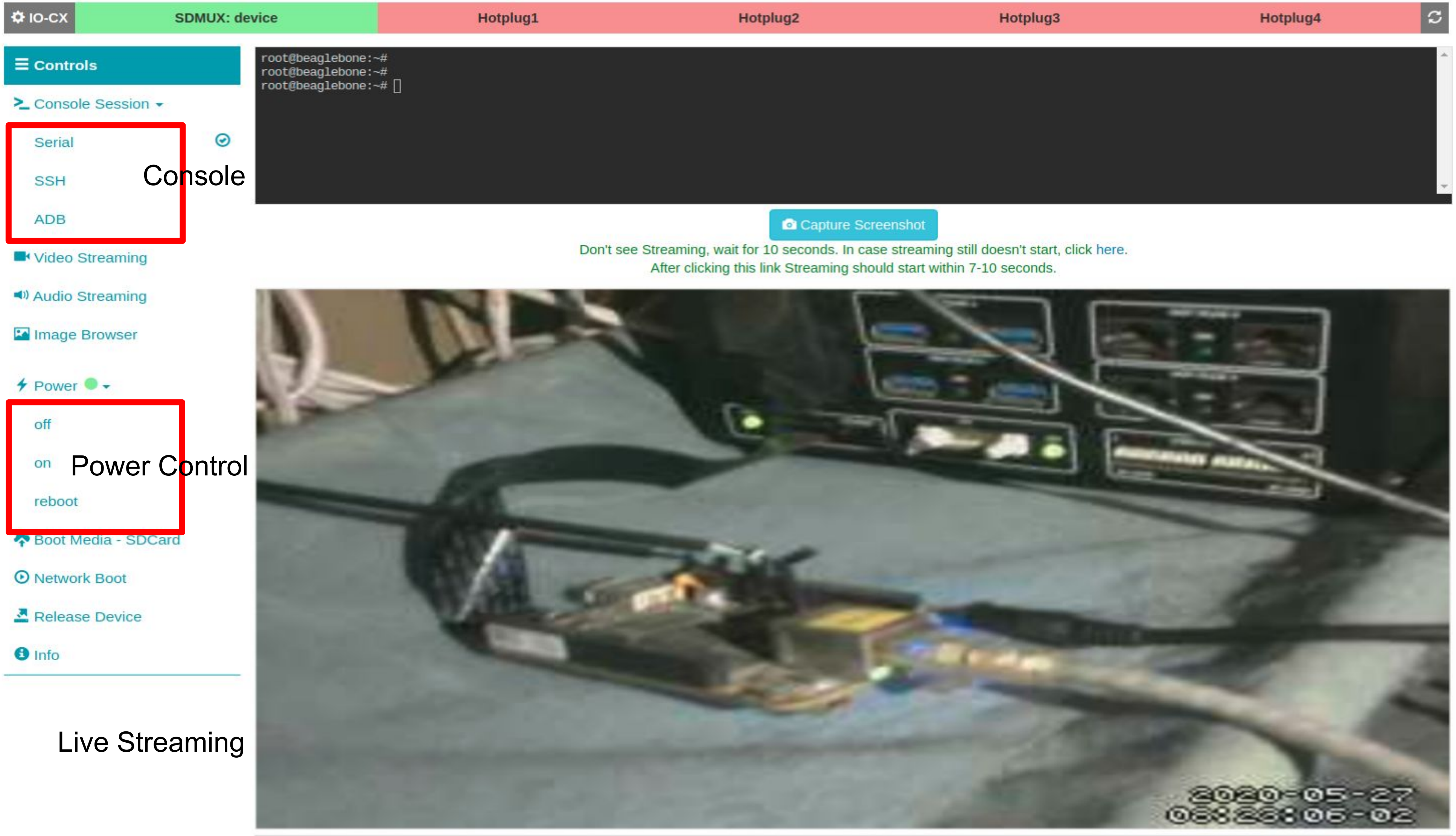
Refresh IOCX Status

Show 50 entries — Search

Device Name	Device type	Device status	tags	Zombie Worker	Device Port	Mux Mode	IOCX Connected	Actions
raspbian	RasPi3	Idle		BFZombie5	3	SDMUX	True	SSH NAT

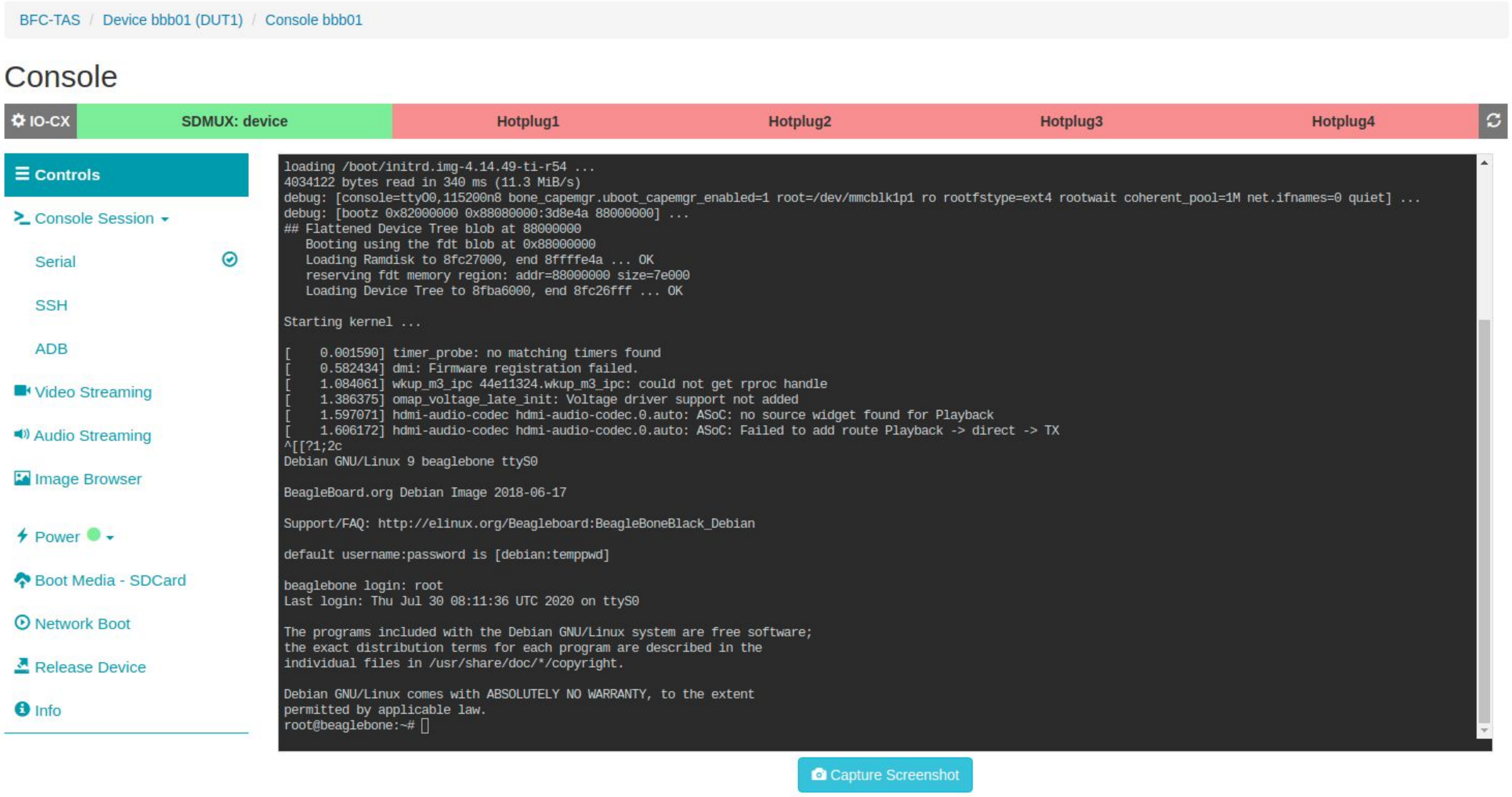
Board/Device Dashboard

- **Device must be allocated to access this page**

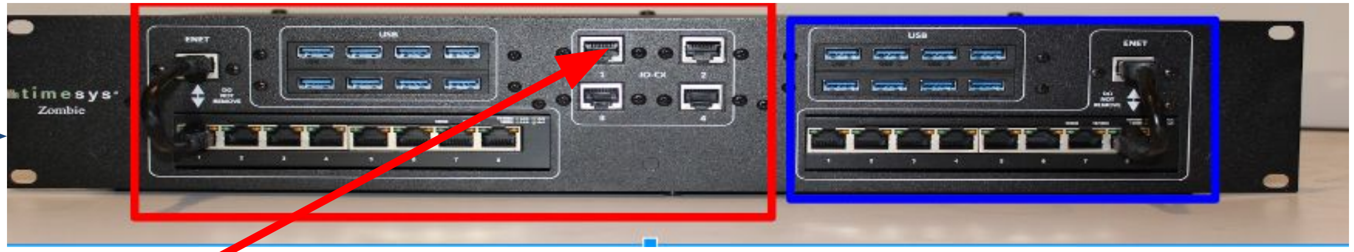


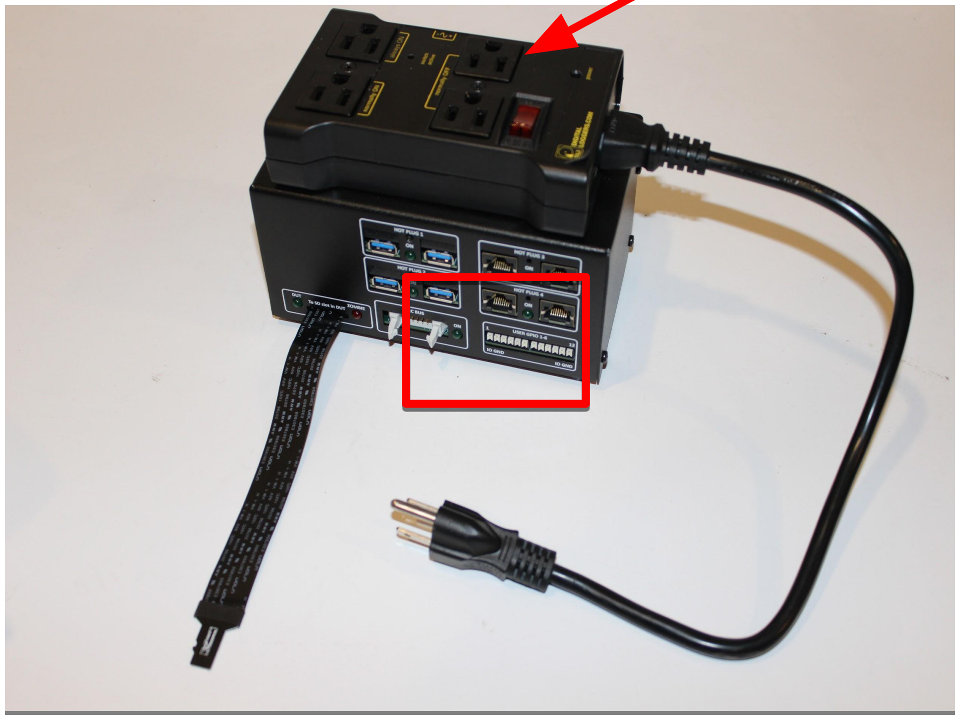
EBF Features

- **Device must be allocated to access this page**
  - **Power Control**
    - Green = ON
    - Red = OFF
  - **New Console Session**
    - Serial
    - SSH
    - adb (Android)
  - **IO-CX Menu**
    - Green = device controlled
    - Red = zombie controlled
  - **SDCard Boot**
  - **Network Boot**
  - **Release Device**



GPIO Connections



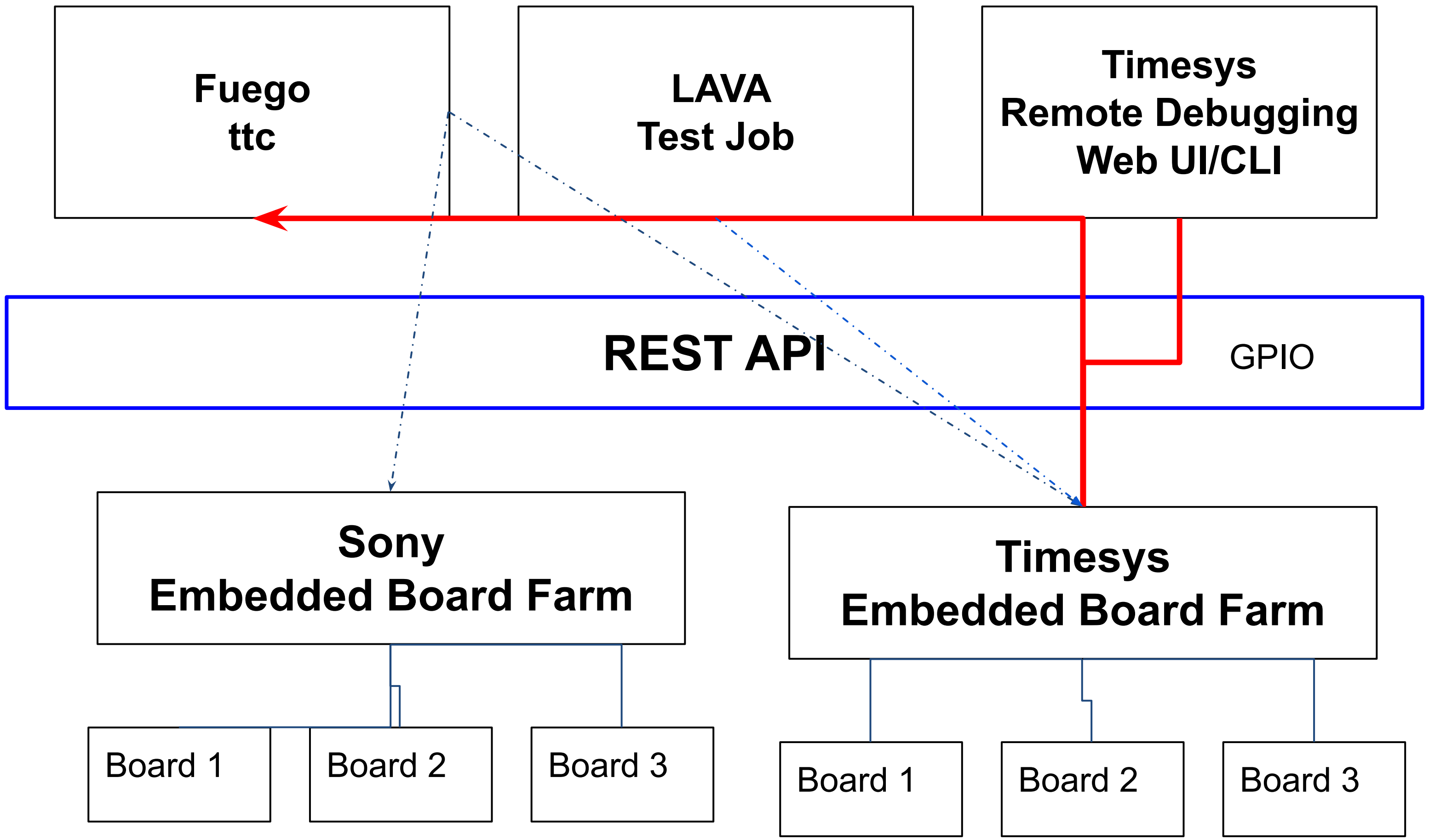


Pins 1-6

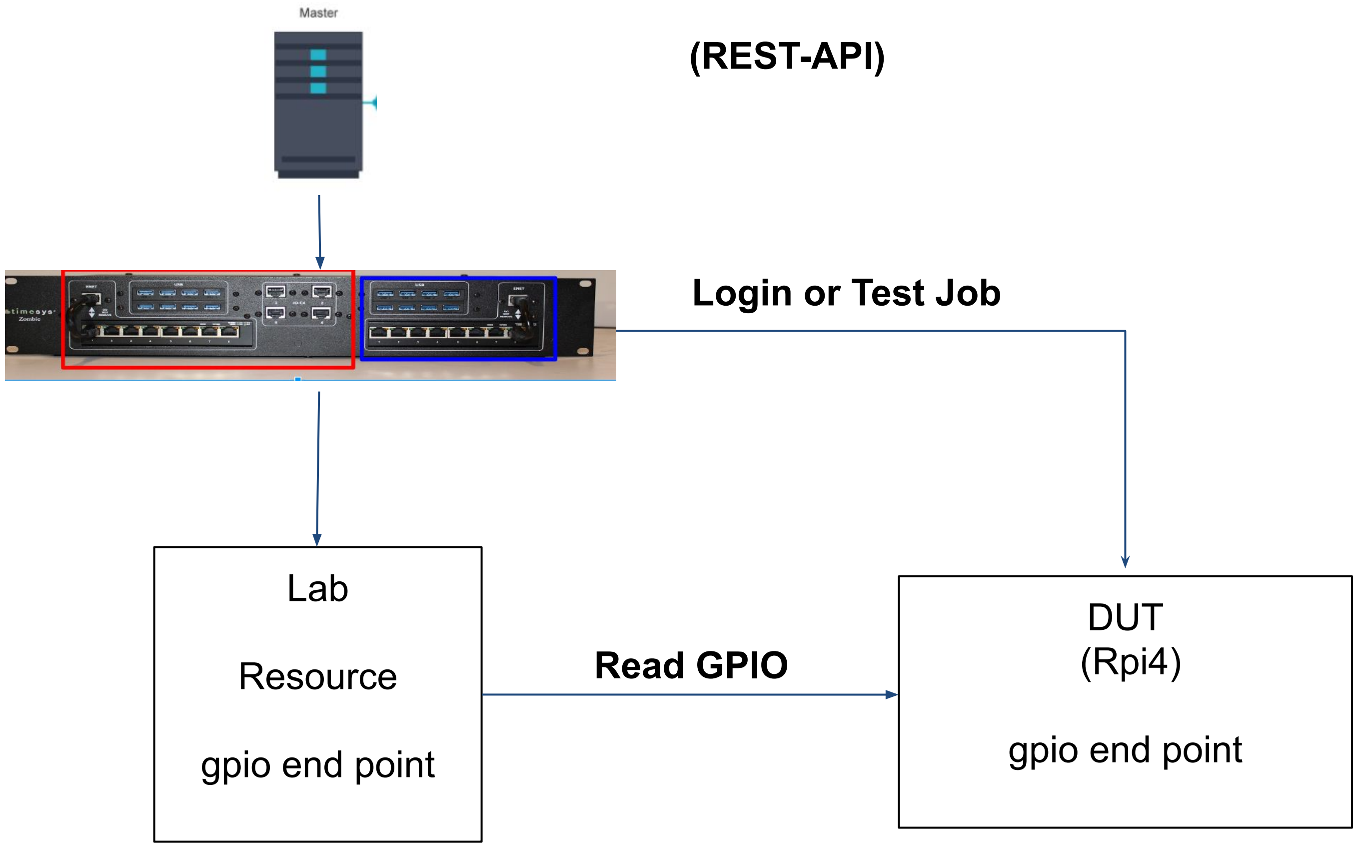
Pins 7-8

Prototype use case:
Lab-independent GPIO test

High Level Concept illustration using real examples



GPIO REST API use case



Assumption:
Lab knows the binding of DUT and the controller - Query the controller ID and types

1. **Manual Execution**
   a. Login to DUT
   b. Set GPIO pins on DUT
   c. Read GPIO from Lab Controller using REST API
2. **Test Script**
   a. Set GPIO pins on DUT
   b. Read from DUT GPIO from Lab Controller using REST API
3. **Test Automation**
   a. Test job to Auto deploy on DUT (Test job include script to set GPIO on DUT and and invoke Lab Controller API
   b. Run Test
   c. Collect results

```
1..5
# running test on bfc.timesys.com:raspi4_gpio
ok 1 - check for /sys/class/gpio
# Lab endpoint prepared to read.
ok 2 - gpio20 directory exists
ok 3 - check that gpio 20 has direction 'out'
# value read from lab endpoint=1
ok 4 - write a 1 to gpio 20
# value read from lab endpoint=0
ok 5 - write a 0 to gpio 20
============================
SUCCESS
```

Pipeline

```
Closing test_set udt.
+ set +x
<LAVA_SIGNAL_ENDRUN 1_test-job 1607_1.1.4.5>
Received signal: <ENDRUN> 1_test-job 1607_1.1.4.5
Ending use of test pattern.
Ending test lava.1_test-job (1607_1.1.4.5), duration 1.02
case: 1_test-job
definition: lava
duration: 1.03
namespace: common
path: inline/test-job.yaml
metadata: [object Object]
run: [object Object]
result: pass
revision: unspecified
uuid: 1607_1.1.4.5
<LAVA_TEST_RUNNER EXIT>
ok: lava_test_shell seems to have completed
GPIO_Test:
  result: pass
  set: udt.
end: 3.1 lava-test-shell (duration 00:00:07) [common]
end: 3 lava-test-retry (duration 00:00:07) [common]
start: 4 finalize (timeout 10:00:00) [common]
start: 4.1 power-off (timeout 00:01:00) [common]
end: 4.1 power-off (duration 00:00:00) [common]
case: power-off
definition: lava
duration: 0.00
extra: ...
level: 4.1
namespace: common
result: pass
start: 4.2 read-feedback (timeout 10:00:00) [common]
Listened to connection for namespace 'common' for 1s
Finalising connection for namespace 'common'
logout
root@raspberrypi:~# logout
end: 4.2 read-feedback (duration 00:00:01) [common]
end: 4 finalize (duration 00:00:01) [common]
Cleaning after the job
Root tmp directory removed at /var/lib/lava/dispatcher/tmp/1607
Job finished correctly
case: job
definition: lava
result: pass
```

Top of page

REST API Details

GPIO REST API details

http://{EBF IP Address}/api/<DeviceName>/gpio/<command>/<gpio_pin_pattern(location)>/<gpio_pin_data>

Result Format:

{“result”: “success”, “data”: <API dependent>}
{“result”: “fail”, "message": ”<reason for failure>”}

command	gpio_pin_pattern	gpio_pin_data (optional)	Examples
set_mode note: 'mode' refers to read or write	Lab Pin #(decimal)	{read /write }	command: set_mode 6 “data”: write output: {“result”: “success”, “data”:”write”}
get_mode	Lab Pin #(decimal)		command: get_mode 6 output: {“result”: “success”, “data”: “read”} note: 0 is considered 'write', and 1 is considered 'read' for Timesys lab controller
write	Lab Pin #(decimal)	0 or 1	command: write 6 0 output: {“result”: “success”, “data”: “0”}
read	Lab Pin #(decimal)		command: read 6 output: {“result”: “success”, “data”: “0”}
set_mode_mask	Lab pin locations pattern mask	0-255 (for a 8 pin controller)	command: set_mode_mask 255 170 output: {“result”: “success”, “data”: “170”}
get_mode_mask	Lab pin locations pattern mask		command: get_mode_mask 255 output: {“result”: “success”, “data”: “170”}
write_mask	Lab pin locations pattern mask	0-255	command:write_mask 255 42 output: {“result”: “success”, “data”: “42”}
read_mask	Lab pin locations pattern mask		command: read_mask 255 output: {“result”: “success”, “data”: “42”}

What the API looks like in practice

Excerpt from gpio_test.sh:

```
# DUT Pin 20 is set in DUT_GPIO_NUM
# Lab Controller GPIO Pin 6 is set in LAB_GPIO_NUM and connected to DUT Pin20
test_desc4="write a 1 to gpio $DUT_GPIO_NUM"

# write to the DUT GPIO using sysfs
echo 1 >/sys/class/gpio/gpio${DUT_GPIO_NUM}/value

# read lab controller
URL=https://${LAB_SERVER}/api/${BOARD_NAME}/gpio/read/${LAB_GPIO_NUM}
value=$(wget -q -O- $URL | jq '.data')
echo "# value read from lab endpoint=$value"

if [ "$value" = 1 ] ; then
    echo "ok 4 - $test_desc4“
else
    echo “not ok 4 - $test_desc4”
fi
```

Comparison of command line and REST API

	Function	CLI command	REST API
01	List Devices	ebf list devices	http://{lab server}/api/devices/
02	Allocate Device	ebf <Device Name> allocate	http://{lab server}/api/devices/<DeviceName>/assign/
03	Device Power ON	ebf <Device Name> power on	http://{lab server}/api/devices/<DeviceName>/power/on/
04	Device Power Status	ebf <Device Name> power status	http://{lab server}/api/devices/<DeviceName>/power/
05	Device Power OFF	ebf <Device Name> power off	http://{lab server}/api/devices/<DeviceName>/power/off/

EBF CLI is implemented using the REST API

Wrap-up

Issues Encountered

- Differentiating Test Framework interface from test interface
  - Some actions are performed by the Test Framework:
    - run, upload, download
  - Some actions are performed by the test:
    - gpio operations (set direction, read, write)
  - Different frameworks put control of operations in different places
- Determining pre-defined data vs. discovered data
  - Example: Currently hardcode GPIO numbers for DUT and lab endpoint
    - Would be better to discover mapping between them
    - Will be different per lab (depends on wiring)
- Supporting full range of operations:
  - Fuego needs recursive file copy, but REST API only supports single file
    - Worked around the issue, but need to decide exact features for API
      - ie – Refine the API
- Needs integration with larger CI loop

What’s next

- Have demonstrated basic concept
- Need to create APIs for other lab resource types (other endpoints)
  - Pretty sure many resources will use “start capture”, “end capture”, and "get_log" actions
    - e.g. power measurement, audio capture, video capture
  - Decide resource-specific actions to support
    - e.g For a power measurement resource, only support “get_log”, or support aggregate operations, like “get_max_power”?
- Run different tests, and see what issues crop up

What’s next

- Have demonstrated basic concept
- Need to create APIs for other lab resource types (other endpoints)
  - Pretty sure many resources will use “start capture” and “end capture” actions
    - e.g. power measurement, audio capture, video capture
  - Decide resource-specific actions to support
    - e.g For a power measurement resource, only support “get_log”, or support aggregate operations, like “get_max_power”?
- Run different tests, and see what issues crop up
- Convince other labs and frameworks to adopt API

What's next

- Have demonstrated basic concept
- Need to create APIs for other lab resource types (other endpoints)
  - Pretty sure many resources will use "start capture" and "end capture" actions
    - e.g. power measurement, audio capture, video capture
  - Decide resource-specific actions to support
    - e.g For a power measurement resource, only support "get_log", or support aggregate operations, like "get_max_power"?
- Run different tests, and see what issues crop up
- Convince other labs and frameworks to adopt API
  - Start sharing tests
  - Profit from a community of tests and results!

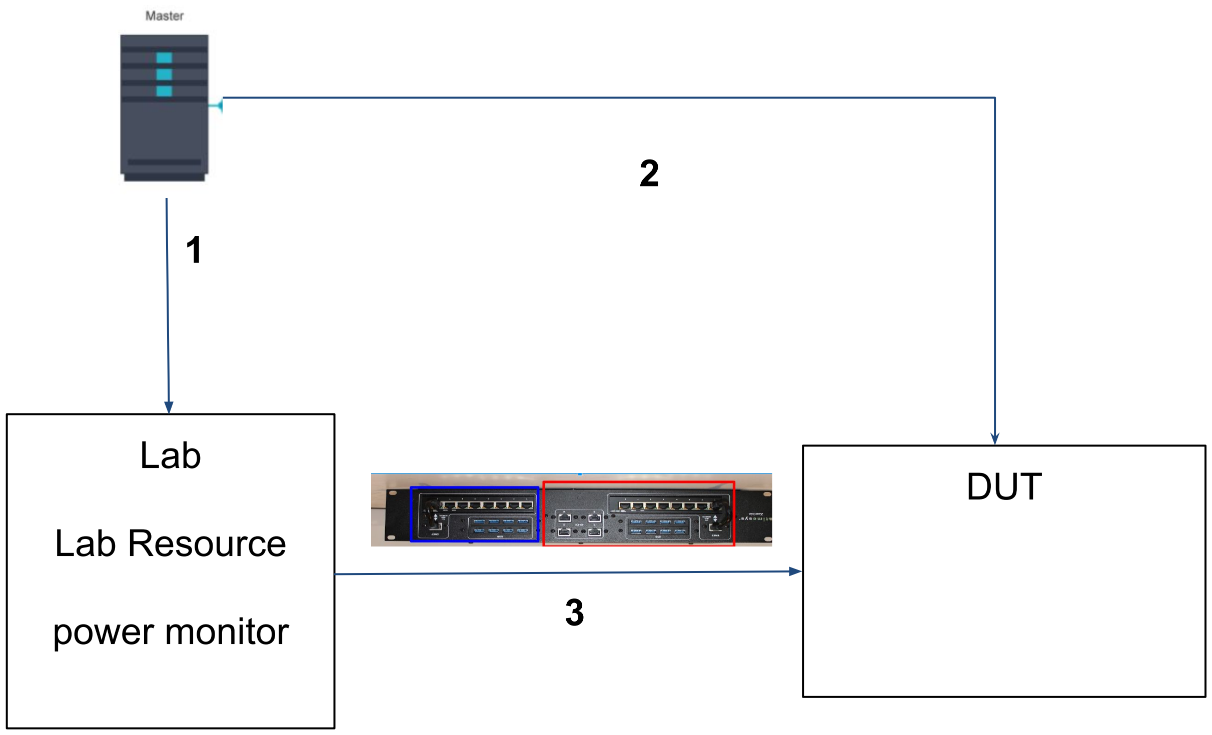
GitHub Repository

https://github.com/TimesysGit/board-farm-rest-api

Questions or Comments?

Additional Use Case Examples

Power measurement



Power measuring lab end point

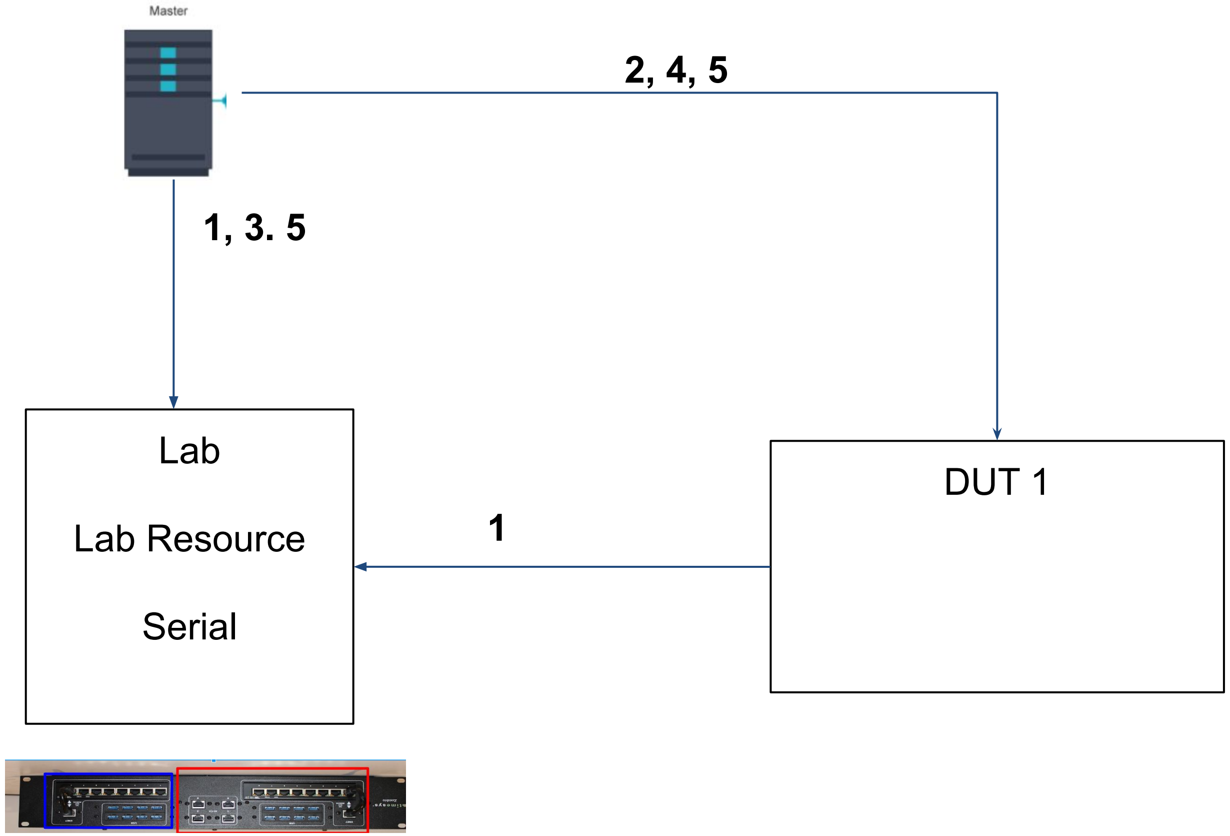
Test of power drawn during test load

1. Use REST API to control the lab resource end point
   a) start measuring
2. Start DUT test load
3. Use REST API to control the lab resource end point
   a) stop measuring
   b) collect the results

Test or framework can analyze power log for test pass/fail condition

Analysis does not need to be done on DUT

Serial



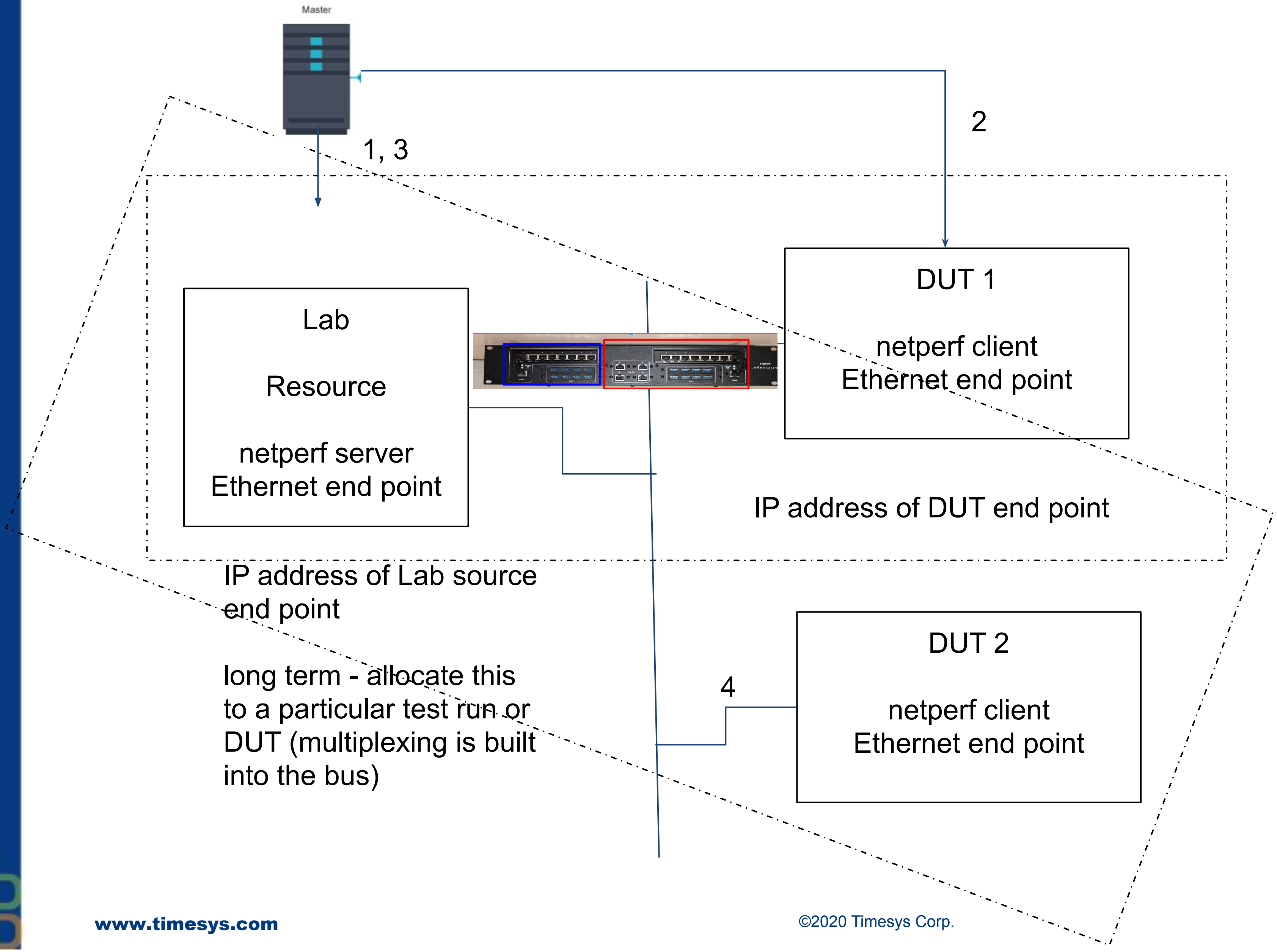
Test of Serial hardware

RS232

1. Use REST API to configure lab resource as Rx or Tx, and baud rate
2. Use local commands to set DUT serial RX or TX and baud rate
3. Initiate capture
4. Initiate transmission
5. End capture, collect log
6. Compare transmission vs capture data

Can also test RS485 (multi drop)

Multiplexed or Dynamic resources use case



Test Network Performance

1. REST API to control lab resource endpoint
   1. start netperf
   2. reserve for use by DUT 1
2. Start netperf client
   1. communicate server endpoint address
   2. collect log
3. REST API to control the lab resource end point
   1. release or stop netperf server
4. Can reassign netperf server to a different DUT for a subsequent test
   1. i.e, the resource is multiplexed between DUTs