labgrid Documentation

Release 0.1.1.dev61+g49c2d29

Jan Luebbe, Rouven Czerwinski

Aug 10, 2017

Contents

Labgrid is a embedded board control python library with a focus on testing, development and general automation. It includes a remote control layer to control boards connected to other hosts.

The idea behind labgrid is to create an abstraction of the hardware control layer needed for testing of embedded systems, automatic software installation and automation during development. Labgrid itself is *not* a testing framework, but is intended to be combined with pytest (and additional pytest plugins). Please see *Design Decisions* for more background information.

It currently supports:

- pytest plugin to write tests for embedded systems connecting serial console or SSH
- remote client-exporter-coordinator infrastructure to make boards available from different computers on a network
- power/reset management via drivers for power switches or onewire PIOs
- upload of binaries via USB: imxusbloader/mxsusbloader (bootloader) or fastboot (kernel)
- functions to control external services such as emulated USB-Sticks and the hawkBit deployment service

While labgrid is currently used for daily development on embedded boards and for automated testing, several planned features are not yet implemented and the APIs may be changed as more use-cases appear. We appreciate code contributions and feedback on using labgrid on other environments (see *Contributing* for details). Please consider contacting us (via a GitHub issue) before starting larger changes, so we can discuss design trade-offs early and avoid redundant work. You can also look at *Ideas* for enhancements which are not yet implemented.

CHAPTER 1

Getting Started

This section of the manual contains introductory tutorials for installing labgrid, running your first test and setting up the distributed infrastructure.

Running Your First Test

In many cases, the easiest way is to install labgrid into a virtualenv:

```
$ virtualenv -p python3 labgrid-venv
$ source labgrid-venv/bin/activate
```

Start by installing labgrid, either by running:

```
$ pip install labgrid
```

or by cloning the repository and installing manually:

```
$ git clone https://github.com/labgrid-project/labgrid
$ cd labgrid && python3 setup.py install
```

Test your installation by running:

```
$ labgrid-client --help
usage: labgrid-client [-h] [-x URL] [-c CONFIG] [-p PLACE] [-d] COMMAND ...
...
```

If the help for labgrid-client does not show up, open an Issue. If everything was successful so far, start by copying the initial example:

```
$ mkdir ../first_test/
$ cp examples/shell/* ../first_test/
$ cd ../first_test/
```

Connect your embedded board (raspberry pi, riotboard, ...) to your computer and adjust the `port` parameter of the `RawSerialPort` resource and `username` and `password` of the ShellDriver driver in `local.yaml`:

```
targets:
  main:
    resources:
      RawSerialPort:
        port: "/dev/ttyUSB0"
    drivers:
      ManualPowerDriver:
        name: "example"
      SerialDriver: {}
      ShellDriver:
        prompt: 'root@\w+:[^ ]+ '
        login_prompt: ' login: '
        username: 'root'
```

You can check which device name gets assigned to your USB-Serial converter by unplugging the converter, running `dmesg -w` and plugging it back in. Boot up your board (manually) and run your first test:

```
$ pytest --lg-env local.yaml test_shell.py
```

It should return successfully, in case it does not, open an Issue.

Setting Up the Distributed Infrastructure

The labgrid distributed infrastructure consists of three components:

1. Coordinator
2. Exporter
3. Client

The system needs at least one coordinator and exporter, these can run on the same machine. The client is used to access functionality provided by an exporter. Over the course of this tutorial we will set up a coordinator and exporter, and learn how to access the exporter via the client.

Coordinator

To start the coordinator, we will download labgrid and select the `coordinator` extra. You can reuse the virtualenv created in the previous section.

```
$ git clone https://github.com/labgrid-project/labgrid
$ cd labgrid && pip install labgrid[coordinator]
```

All necessary dependencies should be installed now, we can start the coordinator by running `crossbar start` inside of the repository.

Note: This is possible because the labgrid repository contains the crossbar configuration the coordinator in the `.crossbar` folder.

Exporter

The exporter needs a configuration file written in YAML syntax, listing the resources to be exported from the local machine. The config file contains one or more named resource groups. Each group contains one or more resource declarations and optionally a location string (see the configuration reference for details).

For example, to export a `RawSerialPort` with the group name *example-port* and the location *example-location*:

```
example-group:
  location: example-location
  RawSerialPort:
    port: /dev/ttyUSB0
```

The exporter can now be started by running:

```
$ labgrid-exporter configuration.yaml
```

Additional groups and resources can be added:

```
example-group:
  location: example-location
  RawSerialPort:
    port: /dev/ttyUSB0
  NetworkPowerPort:
    model: netio
    host: netio1
    index: 3
example-group-2:
  RawSerialPort:
    port: /dev/ttyUSB1
```

Restart the exporter to activate the new configuration.

Client

Finally we can test the client functionality, run:

```
$ labgrid-client resources
kiwi/example-group/NetworkPowerPort
kiwi/example-group/RawSerialPort
kiwi/example-group-2/RawSerialPort
```

You can see the available resources listed by the coordinator. The groups *example-group* and *example-group-2* should be available there.

To show more details on the exported resources, use `-v` (or `-vv`):

```
$ labgrid-client resources -v
Exporter 'kiwi':
  Group 'example-group' (kiwi/example-group/*):
    Resource 'NetworkPowerPort' (kiwi/example-group/NetworkPowerPort[/
↪NetworkPowerPort]):
      {'acquired': None,
       'avail': True,
       'cls': 'NetworkPowerPort',
       'params': {'host': 'netio1', 'index': 3, 'model': 'netio'}}
...
```

You can now add a place with:

```
$ labgrid-client --place example-place create
```

And add resources to this place (`-p` is short for `--place`):

```
$ labgrid-client -p example-place add-match */example-port/*
```

Which adds the previously defined resource from the exporter to the place. To interact with this place, it needs to be acquired first, this is done by

```
$ labgrid-client -p example-place acquire
```

Now we can connect to the serial console:

```
$ labgrid-client -p example-place console
```

For a complete reference have a look at the `labgrid-client(1)` man page.

udev Matching

Labgrid allows the exporter (or the client-side environment) to match resources via udev rules. The udev resources become available to the test/exporter as soon es they are plugged into the computer, e.g. allowing an exporter to export all USB ports on a specific hub and making a `NetworkSerialPort` available as soon as it is plugged into one of the hub's ports. The information udev has on a device can be viewed by executing:

```
$ udevadm info /dev/ttyUSB0
...
E: ID_MODEL_FROM_DATABASE=CP210x UART Bridge / myAVR mySmartUSB light
E: ID_MODEL_ID=ea60
E: ID_PATH=pci-0000:00:14.0-usb-0:5:1.0
E: ID_PATH_TAG=pci-0000_00_14_0-usb-0_5_1_0
E: ID_REVISION=0100
E: ID_SERIAL=Silicon_Labs_CP2102_USB_to_UART_Bridge_Controller_P-00-00682
E: ID_SERIAL_SHORT=P-00-00682
E: ID_TYPE=generic
...
```

In this case the device has an `ID_SERIAL_SHORT` key with a unique ID embedded in the USB-serial converter. The resource match configuration for this USB serial converter is:

```
USBSerialPort:
  match:
    'ID_SERIAL_SHORT': 'P-00-00682'
```

This section can now be added under the resource key in an environment configuration or under its own entry in an exporter configuration file.

Using a Strategy

Strategies allow the labgrid library to automatically bring the board into a defined state, e.g. boot through the bootloader into the Linux kernel and log in to a shell. They have a few requirements:

- A driver implementing the `PowerProtocol`, if no controllable infrastructure is available a `ManualPowerDriver` can be used.
- A driver implementing the `LinuxBootProtocol`, usually a specific driver for the board's bootloader
- A driver implementing the `CommandProtocol`, usually a `ShellDriver` with a `SerialDriver` below it.

Labgrid ships with two builtin strategies, `BareboxStrategy` and `UBootStrategy`. These can be used as a reference example for simple strategies, more complex tests usually require the implementation of your own strategies.

To use a strategy, add it and its dependencies to your configuration YAML, retrieve it in your test and call the `transition(status)` function.

::

```
>>> strategy = target.get_driver(strategy)
>>> strategy.transition("barebox")
```

An example using the pytest plugin is provided under *examples/strategy*.

CHAPTER 2

Overview

Installation

The default installation is available via PyPI:

```
$ pip install labgrid
```

or by cloning the repository and installing manually:

```
$ git clone https://github.com/labgrid-project/labgrid
$ cd labgrid && python3 setup.py install
```

Extra Requirements

Labgrid supports different extras:

- onewire: install onewire support, requires `onewire>=0.0.2` from PyPI and additionally `libow-dev` on Debian based distributions.
- coordinator: installs required depencies to start a crossbar coordinator

The extras can be selected by passing them after the package name in square brackets:

```
$ pip install labgrid[onewire]
```

or to enable both:

```
$ pip install labgrid[onewire,coordinator]
```

Depending on the used shell settings, the brackets may have to be escaped via `\`.

Architecture

Labgrid can be used in several ways:

- on the command line to control individual embedded systems during development ("board farm")
- via a pytest plugin to automate testing of embedded systems
- as a python library in other programs

In the labgrid library, a controllable embedded system is represented as a `Target`. *Targets* normally have several `Resource` and `Driver` objects, which are used to store the board-specific information and to implement actions on different abstraction levels. For cases where a board needs to be transitioned to specific states (such as *off*, *in bootloader*, *in Linux shell*), a `Strategy` (a special kind of *Driver*) can be added to the *Target*.

While labgrid comes with implementations for some resources, drivers and strategies, custom implementations for these can be registered at runtime. It is expected that for complex use-cases, the user would implement and register a custom *Strategy* and possibly some higher-level *Drivers*.

Resources

Resources are passive and only store the information to access the corresponding part of the *Target*. Typical examples of resources are `RawSerialPort`, `NetworkPowerPort` and `AndroidFastboot`.

An important type of *Resources* are `ManagedResources`. While normal *Resources* are always considered available for use and have fixed properties (such as the `/dev/ttyUSB0` device name for a `RawSerialPort`), the *ManagedResources* are used to represent interfaces which are discoverable in some way. They can appear/disappear at runtime and have different properties each time they are discovered. The most common examples of *ManagedResources* are the various USB resources discovered using udev, such as `USBSerialPort`, `IMXUSBLoader` or `AndroidFastboot`.

Drivers and Protocols

A labgrid `Driver` uses one (or more) *Resources* and/or other, lower-level *Drivers* to perform a set of actions on a *Target*. For example, the `NetworkPowerDriver` uses a `NetworkPowerPort` resource to control the *Target's* power supply. In this case, the actions are "on", "off", "cycle" and "get".

As another example, the `ShellDriver` uses any driver implementing the `ConsoleProtocol` (such as a `SerialDriver`, see below). The *ConsoleProtocol* allows the *ShellDriver* to work with any specific method of accessing the board's console (locally via USB, over the network using a console server or even an external program). At the *ConsoleProtocol* level, characters are sent to and received from the target, but they are not yet interpreted as specific commands or their output.

The *ShellDriver* implements the higher-level `CommandProtocol`, providing actions such as "run" or "run_check". Internally, it interacts with the Linux shell on the target board. For example, it:

- waits for the login prompt
- enters user name and password
- runs the requested shell command (delimited by marker strings)
- parses the output
- retrieves the exit status

Other drivers, such as the `SSHDriver`, also implement the *CommandProtocol*. This way, higher-level code (such as a test suite), can be independent of the concrete control method on a given board.

Binding and Activation

When a *Target* is configured, each driver is "bound" to the resources (or other drivers) required by it. Each *Driver* class has a "bindings" attribute, which declares which *Resources* or *Protocols* it needs and under which name they should be available to the *Driver* instance. The binding resolution is handled by the *Target* during the initial configuration and results in a directed, acyclic graph of resources and drivers. During the lifetime of a *Target*, the bindings are considered static.

In most non-trivial target configurations, some drivers are mutually exclusive. For example, a *Target* may have both a `ShellDriver` and a `BareboxDriver`. Both bind to a driver implementing the *ConsoleProtocol* and provide the *CommandProtocol*. Obviously, the board cannot be in the bootloader and in Linux at the same time, which is represented in labgrid via the `BindingState` (*bound*/*active*). If, during activation of a driver, any other driver in its bindings is not active, they will be activated as well.

Activating and deactivating *Drivers* is also used to handle *ManagedResources* becoming available/unavailable at runtime. If some resources bound to by the activating drivers are currently unavailable, the *Target* will wait for them to appear (with a per resource timeout). A realistic sequence of activation might look like this:

- enable power (`PowerProtocol.on`)
- activate the `IMXUSBDriver` driver on the target (this will wait for the `IMXUSBLoader` resource to be available)
- load the bootloader (`BootstrapProtocol.load`)
- activate the `AndroidFastbootDriver` driver on the target (this will wait for the `AndroidFastboot` resource to be available)
- boot the kernel (`AndroidFastbootDriver.boot`)
- activate the `ShellDriver` driver on the target (this will wait for the `USBSerialPort` resource to be available and log in)

Any *ManagedResources* which become unavailable at runtime will automatically deactivate the dependent drivers.

Strategies

Especially when using labgrid from pytest, explicitly controlling the board's boot process can distract from the individual test case. Each `Strategy` implements the board- or project-specific actions necessary to transition from one state to another. Labgrid includes the `BareboxStrategy` and the `UBootStrategy`, which can be used as-is for simple cases or serve as an example for implementing a custom strategy.

Strategies themselves are not activated/deactivated. Instead, they control the states of the other drivers explicitly and execute actions to bring the target into the requested state.

See the strategy example (`examples/strategy`) and the included strategies in `labgrid/strategy` for some more information.

For more information on the reasons behind labgrid's architecture, see *Design Decisions*.

Remote Resources and Places

Labgrid contains components for accessing resources which are not directly accessible on the local machine. The main parts of this are:

labgrid-coordinator (crossbar component) Clients and exporters connect to the coordinator to publish resources, manage place configuration and handle mutual exclusion.

labgrid-exporter **(CLI)** Exports explicitly configured local resources to the coordinator and monitors these for changes in availability or parameters.

labgrid-client **(CLI)** Configures places (consisting of exported resources) and allows command line access to some actions (such as power control, bootstrap, fastboot and the console).

RemotePlace (managed resource) When used in a *Target*, the RemotePlace expands to the resources configured for the named places.

These components communicate over the WAMP implementation Autobahn and the Crossbar WAMP router.

Coordinator

The *Coordinator* is implemented as a Crossbar component and is started by the router. It provides separate RPC methods for the exporters and clients.

The coordinator keeps a list of all resources for clients and notifies them of changes as they occur. The resource access from clients does not pass through the coordinator, but is instead done directly from client to exporter, avoiding the need to specify new interfaces for each resource type.

The coordinator also manages the registry of "places". These are used to configure which resources belong together from the user's point of view. A *place* can be a generic rack location, where different boards are connected to a static set of interfaces (resources such as power, network, serial console, ...).

Alternatively, a *place* can also be created for a specific board, for example when special interfaces such as GPIO buttons need to be controlled and they are not available in the generic locations.

Each place can have aliases to simplify accessing a specific board (which might be moved between generic places). It also has a comment, which is used to store a short description of the connected board.

Finally, a place is configured with one or more *resource matches*. A resource match pattern has the format `<exporter>/<group>/<class>`, where each component may be replaced with the wildcard `*`.

Some commonly used match patterns are:

/1001/ Matches all resources in groups named 1001 from all exporters.

***/1001/NetworkPowerPort** Matches only the NetworkPowerPort resource in groups named 1001 from all exporters. This is useful to exclude a NetworkSerialPort in group 1001 in cases where the serial console is connected somewhere else (such as via USB on a different exporter).

exporter1/hub1-port1/* Matches all resources exported from exporter1 in the group hub1-port1. This is an easy way to match several USB resources related to the same board (such as a USB ROM-Loader interface, Android fastboot and a USB serial gadget in Linux).

To avoid conflicting access to the same resources, a place must be *aquired* before it is used and the coordinator also keeps track of which user on which client host has currently acquired the place. The resource matches are only evaluated while a place is being acquired and cannot be changed until it is *released* again.

Exporter

An exporters registers all its configured resources when it connects to the router and updates the resource parameters when they change (such as (dis-)connection of USB devices). Internally, the exporter uses the normal `Resource` (and `ManagedResource`) classes as the rest of labgrid. By using *ManagedResources*, availability and parameters for resources such as USB serial ports are tracked and sent to the coordinator.

For some specific resources (such as `USBSerialPorts`), the exporter uses external tools to allow access by clients (`ser2net` in the serial port case).

Resources which do not need explicit support in the exporter, are just published as declared in the configuration file. This is useful to register externally configured resources such as network power switches or serial port servers with a labgrid coordinator.

Client

The client requests the current lists of resources and places from the coordinator when it connects to it and then registers for change events. Most of its functionality is exposed via the *labgrid-client* CLI tool. It is also used by the `RemotePlace` resource (see below).

Besides viewing the list of *resources*, the client is used to configure and access *places* on the coordinator. For more information on using the CLI, see the manual page for *labgrid-client*.

RemotePlace

To use the resources configured for a *place* to control the corresponding board (whether in pytest or directly with the labgrid library), the `RemotePlace` resource should be used. When a *RemotePlace* is configured for a *Target*, it will create a client connection to the coordinator, create additional resource objects for those configured for that place and keep them updated at runtime.

The additional resource objects can be bound to by drivers as normal and the drivers do not need to be aware that they were provided by the coordinator. For resource types which do not have an existing, network-transparent protocol (such as USB ROM loaders or JTAG interfaces), the driver needs to be aware of the mapping done by the exporter.

For generic USB resources, the exporter for example maps a `AndroidFastboot` resource to a `NetworkAndroidFastboot` resource and adds a hostname property which needs to be used by the client to connect to the exporter. To avoid the need for additional remote access protocols and authentication, labgrid currently expects that the hosts are accessible via SSH and that any file names refer to a shared filesystem (such as NFS or SMB).

Note: Using SSH's session sharing (`ControlMaster auto`, `ControlPersist`, ...) makes *RemotePlaces* easy to use even for exporters with require passwords or more complex login procedures.

For exporters which are not directly accessible via SSH, add the host to your .ssh/config file, with a ProxyCommand when need.

CHAPTER 3

Usage

Library

Labgrid can be used directly as a Python library, without the infrastructure provided by the pytest plugin.

Creating and Configuring Targets

The labgrid library provides two ways to configure targets with resources and drivers: either create the *Target* directly or use *Environment* to load a configuration file.

Targets

At the lower level, a *Target* can be created directly:

```
>>> from labgrid import Target
>>> t = Target('example')
```

Next, the required *Resources* can be created:

```
>>> from labgrid.resource import RawSerialPort
>>> rsp = RawSerialPort(t, port='/dev/ttyUSB0')
```

Then, a *Driver* needs to be created on the *Target*:

```
>>> from labgrid.driver import SerialDriver
>>> sd = SerialDriver(t)
```

As the *SerialDriver* declares a binding to a SerialPort, the target binds it to the resource created above:

```
>>> sd.port
RawSerialPort(target=Target(name='example', env=None), state=<BindingState.active: 2>,
↪ avail=True, port='/dev/ttyUSB0', speed=115200)
```

```
>>> sd.port is rsp
True
```

Before the driver can be used, it needs to be activated:

```
>>> t.activate(sd)
>>> sd.write(b'test')
```

Environments

In practice, is is often useful to separate the *Target* configuration from the code which needs to control the board (such as a test case or installation script). For this use-case, labgrid can construct targets from a configuration file in YAML format:

```
targets:
  example:
    resources:
      RawSerialPort:
        port: '/dev/ttyUSB0'
    drivers:
      SerialDriver: {}
```

To parse this configuration file, use the *Environment* class:

```
>>> from labgrid import Environment
>>> env = Environment('example-env.yaml')
```

Using *Environment.get_target*, the configured *Targets* can be retrieved by name. Without an argument, *get_target* would default to 'main':

```
>>> t = env.get_target('example')
```

To access the target's console, the correct driver object can be found by using *Target.get_driver*:

```
>>> from labgrid.protocol import ConsoleProtocol
>>> cp = t.get_driver(ConsoleProtocol)
>>> cp
SerialDriver(target=Target(name='example', env=Environment(config_file='example.yaml
↪')), state=<BindingState.active: 2>)
>>> cp.write(b'test')
```

When using the `get_driver` method, the driver is automatically activated. The driver activation will also wait for unavailable resources when needed.

pytest Plugin

Labgrid includes a pytest plugin to simplify writing tests which involve embedded boards. The plugin is configured by providing an environment config file (via the –lg-env pytest option) and automatically creates the targets described in the environment.

Two pytest fixtures are provided:

env (session scope) Used to access the *Environment* object created from the configuration file. This is mostly used for defining custom fixtures at the test suite level.

target (session scope) Used to access the 'main' *Target* defined in the configuration file.

Simple Example

As a minimal example, we have a target connected via a USB serial converter ('/dev/ttyUSB0') and booted to the Linux shell. The following environment config file (`shell-example.yaml`) describes how to access this board:

```
targets:
  main:
    resources:
      RawSerialPort:
        port: '/dev/ttyUSB0'
    drivers:
      SerialDriver: {}
      ShellDriver:
        prompt: 'root@\w+:[^ ]+ '
        login_prompt: ' login: '
        username: 'root'
```

We then add the following test in a file called `test_example.py`:

```
from labgrid.protocol import CommandProtocol

def test_echo(target):
    command = t.get_driver(CommandProtocol)
    result = command.run_check('echo OK')
    assert 'OK' in result
```

To run this test, we simply execute pytest in the same directory with the environment config:

```
$ pytest --lg-env shell-example.yaml --verbose
============================= test session starts ==============================
platform linux -- Python 3.5.3, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
...
collected 1 items

test_example.py::test_echo PASSED
========================== 1 passed in 0.51 seconds ===========================
```

pytest has automatically found the test case and executed it on the target.

Custom Fixture Example

When writing many test cases which use the same driver, we can get rid of some common code by wrapping the *CommandProtocol* in a fixture. As pytest always executes the `conftest.py` file in the test suite directory, we can define additional fixtures there:

```
import pytest

from labgrid.protocol import CommandProtocol

@pytest.fixture(scope='session')
def command(target):
    return target.get_driver(CommandProtocol)
```

With this fixture, we can simplify the `test_example.py` file to:

```
def test_echo(command):
    result = command.run_check('echo OK')
    assert 'OK' in result
```

Strategy Fixture Example

When using a *Strategy* to transition the target between states, it is useful to define a function scope fixture per state in `conftest.py`:

```
import pytest

from labgrid.protocol import CommandProtocol
from labgrid.strategy import BareboxStrategy

@pytest.fixture(scope='session')
def strategy(target):
    try:
        return target.get_driver(BareboxStrategy)
    except NoDriverFoundError:
        pytest.skip("strategy not found")

@pytest.fixture(scope='function')
def bootloader_command(target, strategy, capsys):
    with capsys.disabled():
        strategy.transition('barebox')
    return target.get_active_driver(CommandProtocol)

@pytest.fixture(scope='function')
def shell_command(target, strategy, capsys):
    with capsys.disabled():
        strategy.transition('shell')
    return target.get_active_driver(CommandProtocol)
```

Note: The `capsys.disabled()` context manager is only needed when using the *ManualPowerDriver*, as it will not be able to access the console otherwise. See the corresponding pytest documentation for details.

With the fixtures defined above, switching between bootloader and Linux shells is easy:

```
def test_barebox_initial(bootloader_command):
    stdout = bootloader_command.run_check('version')
    assert 'barebox' in '\n'.join(stdout)

def test_shell(shell_command):
    stdout = shell_command.run_check('cat /proc/version')
    assert 'Linux' in stdout[0]

def test_barebox_after_reboot(bootloader_command):
    bootloader_command.run_check('true')
```

Note: The *bootloader_command* and *shell_command* fixtures use *Target.get_active_driver* to get the currently active *CommandProtocol* driver (either *BareboxDriver* or *ShellDriver*). Activation and deactivation

of drivers is handled by the *BareboxStrategy* in this example.

The *Strategy* needs additional drivers to control the target. Adapt the following environment config file (`strategy-example.yaml`) to your setup:

```
targets:
  main:
    resources:
      RawSerialPort:
        port: '/dev/ttyUSB0'
    drivers:
      ManualPowerDriver:
        name: 'example-board'
      SerialDriver: {}
      BareboxDriver:
        prompt: 'barebox@[^:]+:[^ ]+ '
      ShellDriver:
        prompt: 'root@\w+:[^ ]+ '
        login_prompt: ' login: '
        username: 'root'
      BareboxStrategy: {}
```

For this example, you should get a report similar to this:

```
$ pytest --lg-env strategy-example.yaml -v
============================= test session starts ==============================
platform linux -- Python 3.5.3, pytest-3.0.6, py-1.4.32, pluggy-0.4.0
...
collected 3 items

test_strategy.py::test_barebox_initial
main: CYCLE the target example-board and press enter
PASSED
test_strategy.py::test_shell PASSED
test_strategy.py::test_barebox_after_reboot
main: CYCLE the target example-board and press enter
PASSED

========================== 3 passed in 29.77 seconds ===========================
```

Test Reports

pytest-html

With the pytest-html plugin, the test results can be converted directly to a single-page HTML report:

```
$ pip install pytest-html
$ pytest --lg-env shell-example.yaml --html=report.html
```

JUnit XML

JUnit XML reports can be generated directly by pytest and are especially useful for use in CI systems such as Jenkins with the JUnit Plugin.

They can also be converted to other formats, such as HTML with junit2html tool:

```
$ pip install junit2html
$ pytest --lg-env shell-example.yaml --junit-xml=report.xml
$ junit2html report.xml
```

Labgrid adds additional xml properties to a test run, these are:

- ENV_CONFIG: Name of the configuration file
- TARGETS: List of target names
- TARGET_{NAME}_REMOTE: optional, if the target uses a RemotePlace resource, its name is recorded here
- PATH_{NAME}: optional, labgrid records the name and path
- PATH_{NAME}_GIT_COMMIT: optional, labgrid tries to record git sha1 values for every path
- IMAGE_{NAME}: optional, labgrid records the name and path to the image
- IMAGE_{NAME}_GIT_COMMIT: optional, labgrid tries to record git sha1 values for every image

Command-Line

Labgrid contains some command line tools which are used for remote access to resources. See *labgrid-client*, *labgrid-device-config* and *labgrid-exporter* for more information.

USB stick emulation

Labgrid makes it posible to use a target as an emulated USB stick, allowing upload, modification, plug and unplug events. To use a target as an emulated USB stick, several requirements have to be met:

- OTG support on one of the device USB ports
- losetup from util-linux
- mount from util-linux
- A kernel build with *CONFIG_USB_GADGETFS=m*
- A network connection to the target to use the *SSHDriver* for file uploads

To use USB stick emulation, import `USBStick` from *labgrid.external* and bind it to the desired target:

```
from labgrid.external import USBStick

stick = USBStick(target, '/home/')
```

The above code block creates the stick and uses the */home* directory to store the device images. USBStick images can now be uploaded using the *upload_image* method. Once an image is selected, files can be uploaded and retrived using the *put_file* and *get_file* methods. The *plug_in* and *plug_out* functions plug the emulated USB stick in and out.

hawkBit management API

Labgrid provides an interface to the hawkbit management API. This allows a labgrid test to create targets, rollouts and manage deployments.

```
from labgrid.external import HawkbitTestClient

client = HawkbitTestClient('local', '8080', 'admin', 'admin')
```

The above code connects to a running hawkbit instance on the local computer and uses the default credentials to log in. The *HawkbitTestClient* provides various helper functions to add targets, define distribution sets and assign targets.

CHAPTER 4

Manual Pages

labgrid-client

labgrid-client interface to control boards

Author Rouven Czerwinski <r.czerwinski@pengutronix.de>

organization Labgrid-Project

Date 2017-04-15

Version 0.0.1

Manual section 1

Manual group embedded testing

SYNOPSIS

`labgrid-client --help`

`labgrid-client` -p <place> <command>

`labgrid-client places|resources`

DESCRIPTION

Labgrid is a scalable infrastructure and test architecture for embedded (linux) systems.

This is the client to control a boards status and interface with it on remote machines.

OPTIONS

-h, --help display command line help

-p PLACE, --place PLACE specify the place to operate on

-x, --crossbar-url the crossbar url of the coordinator

-c CONFIG, --config CONFIG set the configuration file

-s STATE, --state STATE set an initial state before executing a command, requires a configuration file and strategy

-d, --debug enable debugging

CONFIGURATION FILE

The configuration file follows the description in `labgrid-device-config`(1).

ENVIRONMENT VARIABLES

Various labgrid-client commands use the following environment variable:

PLACE

This variable can be used to specify a place without using the `-p` option, the `-p` option overrides it.

STATE

This variable can be used to specify a state which the device transitions into before executing a command. Requires a configuration file and a Strategy specified for the device.

MATCHES

Match patterns are used to assign a resource to a specific place. The format is: exporter/group/cls/name, exporter is the name of the exporting machine, group is a name defined within the exporter, cls is the class of the exported resource and name is its name. Wild cards in match patterns are explicitly allowed, * matches anything.

LABGRID-CLIENT COMMANDS

`monitor` Monitor events from the coordinator

`resources (r)` List available resources

`places (p)` List available places

`show` Show a place and related resources

`create` Add a new place (name supplied by -p parameter)

`delete` Delete an existing place

`add-alias` Add an alias to a place

`del-alias` Delete an alias from a place

`set-comment` Update or set the place comment

`add-match` match Add a match pattern to a place, see MATCHES

`del-match` match Delete a match pattern from a place, see MATCHES

`acquire (lock)` Acquire a place

`release (unlock)` Release a place

`env` Generate a labgrid environment file for a place

`power (pw)` action Change (or get) a place's power status, where action is one of get, on, off, status

`console (con)` Connect to the console

`fastboot` Run fastboot

`bootstrap` Start a bootloader

`io` Interact with Onewire devices

EXAMPLES

To retrieve a list of places run:

```
$ labgrid-client places
```

To access a place, it needs to be acquired first, this can be done by running the `acquire command` and passing the placename as a -p parameter:

```
$ labgrid-client -p <placename> acquire
```

Open a console to the acquired place:

```
$ labgrid-client -p <placename> console
```

Add all resources with the group "example-group" to the place example-place:

```
$ labgrid-client -p example-place add-match */example-group/*/*
```

SEE ALSO

`labgrid-exporter`(1)

labgrid-device-config

labgrid test configuration files

Author Rouven Czerwinski <r.czerwinski@pengutronix.de>

organization Labgrid-Project

Date 2017-04-15

Version 0.0.1

Manual section 1

Manual group embedded testing

SYNOPSIS

`*.yaml`

DESCRIPTION

To integrate a device into a labgrid test, labgrid needs to have a description of the device and how to access it.

This manual page is divided into section, each describing one top-level yaml key.

TARGETS

The `targets:` top key configures a `target`, it's `drivers` and `resources`.

The top level key is the name of the target, it needs both a `resources` and `drivers` subkey. The order of instantiated `resources` and `drivers` is important, since they are parsed as an ordered dictionary and may depend on a previous driver.

For a list of available resources and drivers refer to https://labgrid.readthedocs.io/en/latest/configuration.html.

OPTIONS

The `options:` top key configures various options such as the crossbar_url.

KEYS

`crossbar_url` takes as parameter the URL of the crossbar (coordinator) to connect to. Defaults to 'ws://127.0.0.1:20408'.

`crossbar_realm` takes as parameter the realm of the crossbar (coordinator) to connect to. Defaults to 'realm1'.

IMAGES

The `images:` top key provides paths to access preconfigured images to flash onto the board.

KEYS

The subkeys consist of image names as keys and their paths as values. The corresponding name can than be used with the appropriate tool found under TOOLS.

TOOLS

The `tools:` top key provides paths to binaries such as fastboot.

KEYS

fastboot Path to the fastboot binary

mxs-usb-loader Path to the mxs-usb-loader binary

imx-usb-loader Path to the imx-usb-loader binary

IMPORTS

The `imports` key is a list of files which are imported by the environment after loading the configuration. Relative paths to the configuration file are also supported.

EXAMPLES

A sample configuration with one *main* target, accessible via SerialPort */dev/ttyUSB0*, allowing usage of the ShellDriver:

```
targets:
  main:
    resources:
      RawSerialPort:
        port: "/dev/ttyUSB0"
    drivers:
      SerialDriver: {}
      ShellDriver:
        prompt: 'root@\w+:[^ ]+ '
        login_prompt: ' login: '
      username: 'root'
```

SEE ALSO

`labgrid-client`(1), `labgrid-exporter`(1)

labgrid-exporter

labgrid-exporter interface to control boards

Author Rouven Czerwinski <r.czerwinski@pengutronix.de>

organization Labgrid-Project

Date 2017-04-15

Copyright Copyright (C) 2016-2017 Pengutronix. This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Version 0.0.1

Manual section 1

Manual group embedded testing

SYNOPSIS

`labgrid-exporter --help`

`labgrid-exporter *.yaml`

DESCRIPTION

Labgrid is a scalable infrastructure and test architecture for embedded (linux) systems.

This is the man page for the exporter, supporting the export of serial ports, usb tools and various other controllers.

OPTIONS

-h, --help	display command line help
-x, --crossbar-url	the crossbar url of the coordinator
-n, --name	the public name of the exporter

CONFIGURATION

The exporter uses a YAML configuration file which defines groups of releated resources. Furthermore the exporter can start helper binaries such as `ser2net` to export local serial ports over the network.

EXAMPLES

Start the exporter with the configuration file *my-config.yaml*:

```
$ labgrid-exporter my-config.yaml
```

Same as above, but with name `myname`:

```
$ labgrid-exporter -n myname my-config.yaml
```

SEE ALSO

`labgrid-client`(1), `labgrid-device-config`(1)

CHAPTER 5

Configuration

This chapter describes the individual drivers and resources used in a device configuration. Drivers can depend on resources or other drivers, whereas resources have no dependencies.

Here the resource *RawSerialPort* provides the information for the *SerialDriver*, which in turn is needed by the *ShellDriver*. Driver dependency resolution is done by searching for the driver which implements the dependent protocol, all drivers implement one or more protocols.

Resources

Serial Ports

RawSerialPort

A RawSerialPort is a serial port which is identified via the device path on the local computer. Take note that re-plugging USB serial converters can result in a different enumeration order.

```
RawSerialPort:
  port: /dev/ttyUSB0
  speed: 115200
```

The example would access the serial port /dev/ttyUSB0 on the local computer with a baud rate of 115200.

- port (str): path to the serial device
- speed (int): desired baud rate

Used by:

- *SerialDriver*

NetworkSerialPort

A NetworkSerialPort describes a serial port which is exported over the network, usually using RFC2217.

```
NetworkSerialPort:
  host: remote.example.computer
  port: 53867
  speed: 115200
```

The example would access the serial port on computer remote.example.computer via port 53867 and use a baud rate of 115200.

- host (str): hostname of the remote host
- port (str): TCP port on the remote host to connect to
- speed (int): baud rate of the serial port

Used by:

- *SerialDriver*

USBSerialPort

A USBSerialPort describes a serial port which is connected via USB and is identified by matching udev properties. This allows identification through hot-plugging or rebooting.

```
USBSerialPort:
  match:
    'ID_SERIAL_SHORT': 'P-00-00682'
  speed: 115200
```

The example would search for a USB serial converter with the key *ID_SERIAL_SHORT* and the value *P-00-00682* and use it with a baud rate of 115200.

- match (str): key and value for a udev match, see *udev Matching*
- speed (int): baud rate of the serial port

Used by:

- *SerialDriver*

NetworkPowerPort

A NetworkPowerPort describes a remotely switchable power port.

```
NetworkPowerPort:
  model: gude
  host: powerswitch.example.computer
  index: 0
```

The example describes port 0 on the remote power switch *powerswitch.example.computer*, which is a *gude* model.

- model (str): model of the power switch
- host (str): hostname of the power switch
- index (int): number of the port to switch

Used by:

- *NetworkPowerDriver*

NetworkService

A NetworkService describes a remote SSH connection.

```
NetworkService:
  address: example.computer
  username: root
```

The example describes a remote SSH connection to the computer *example.computer* with the username *root*.

- address (str): hostname of the remote system
- username (str): username used by SSH

Used by:

- *SSHDriver*

OneWirePIO

A OneWirePIO describes a onewire programmable I/O pin.

```
OneWirePIO:
  host: example.computer
  path: /29.7D6913000000/PIO.0
```

The example describes a *PIO.0* at device address *29.7D6913000000* via the onewire server on *example.computer*.

- host (str): hostname of the remote system running the onewire server
- path (str): path on the server to the programmable I/O pin

Used by:

- *OneWirePIODriver*

USBMassStorage

A USBMassStorage resource describes a USB memory stick or similar device.

```
USBMassStorage:
  match:
    'ID_PATH': 'pci-0000:06:00.0-usb-0:1.3.2:1.0-scsi-0:0:0:3'
```

- match (str): key and value for a udev match, see *udev Matching*

Used by:

- *USBStorageDriver*

IMXUSBLoader

An IMXUSBLoader resource describes a USB device in the imx loader state.

```
IMXUSBLoader:
  match:
    'ID_PATH': 'pci-0000:06:00.0-usb-0:1.3.2:1.0'
```

- match (str): key and value for a udev match, see *udev Matching*

Used by:

- *IMXUSBDriver*

MXSUSBLoader

An MXSUSBLoader resource describes a USB device in the mxs loader state.

```
MXSUSBLoader:
  match:
    'ID_PATH': 'pci-0000:06:00.0-usb-0:1.3.2:1.0'
```

- match (str): key and value for a udev match, see *udev Matching*

Used by:

- *MXSUSBDriver*

NetworkMXSUSBLoader

A NetworkMXSUSBLoader descibes an *MXSUSBLoader* available on a remote computer.

NetworkIMXUSBLoader

A NetworkIMXUSBLoader descibes an *IMXUSBLoader* available on a remote computer.

AndroidFastboot

An AndroidFastboot resource describes a USB device in the fastboot state.

```
AndroidFastboot:
  match:
    'ID_PATH': 'pci-0000:06:00.0-usb-0:1.3.2:1.0'
```

- match (str): key and value for a udev match, see *udev Matching*

Used by:

- *AndroidFastbootDriver*

USBEthernetInterface

A USBEthernetInterface resource describes a USB device Ethernet adapter.

```
USBEthernetInterface:
  match:
    'ID_PATH': 'pci-0000:06:00.0-usb-0:1.3.2:1.0'
```

- match (str): key and value for a udev match, see *udev Matching*

AlteraUSBBlaster

An AlteraUSBBlaster resource describes an Altera USB blaster.

```
AlteraUSBBlaster:
  match:
    'ID_PATH': 'pci-0000:06:00.0-usb-0:1.3.2:1.0'
```

- match (str): key and value for a udev match, see *udev Matching*

Used by:

- *OpenOCDDriver*

RemotePlace

A RemotePlace describes a set of resources attached to a labgrid remote place.

```
RemotePlace:
  name: example-place
```

The example describes the remote place *example-place*. It will connect to the labgrid remote coordinator, wait until the resources become available and expose them to the internal environment.

- name (str): name or pattern of the remote place

Used by:

- potentially all drivers

udev Matching

udev matching allows labgrid to identify resources via their udev properties. Any udev property key and value can be used, path matching USB devices is allowed as well. This allows exporting a specific USB hub port or the correct identification of a USB serial converter across computers.

The initial matching and monitoring for udev events is handled by the `UdevManager` class. This manager is automatically created when a resource derived from `USBResource` (such as `USBSerialPort`, `IMXUSBLoader` or `AndroidFastboot`) is instantiated.

To identify the kernel device which corresponds to a configured *USBResource*, each existing (and subsequently added) kernel device is matched against the configured resources. This is based on a list of *match entries* which must all be tested successfully against the potential kernel device. Match entries starting with an `@` are checked against the device's parents instead of itself; here one matching parent causes the check to be successful.

A given *USBResource* class has builtin match entries that are checked first, for example that the `SUBSYSTEM` is `tty` as in the case of the *USBSerialPort*. Only if these succeed, match entries provided by the user for the resource instance are considered.

In addition to the properties reported by `udevadm monitor --udev --property`, elements of the `ATTR(S){}` dictionary (as shown by `udevadmin info <device> -a`) are useable as match keys. Finally `sys_name` allows matching against the name of the directory in sysfs. All match entries must succeed for the device to be accepted.

The following examples show how to use the udev matches for some common use-cases.

Matching a USB Serial Converter on a Hub Port

This will match any USB serial converter connected below the hub port 1.2.5.5 on bus 1. The *sys_name* value corresponds to the hierarchy of buses and ports as shown with `lsusb -t` and is also usually displayed in the kernel log messages when new devices are detected.

```
USBSerialPort:
  match:
    '@sys_name': '1-1.2.5.5'
```

Note the `@` in the `@sys_name` match, which applies this match to the device's parents instead of directly to itself. This is necessary for the *USBSerialPort* because we actually want to find the `ttyUSB?` device below the USB serial converter device.

Matching an Android Fastboot Device

In this case, we want to match the USB device on that port directly, so we don't use a parent match.

```
AndroidFastboot:
  match:
    'sys_name': '1-1.2.3'
```

Matching a Specific UART in a Dual-Port Adapter

On this board, the serial console is connected to the second port of an on-board dual-port USB-UART. The board itself is connected to the bus 3 and port path 10.2.2.2. The correct value can be shown by running `udevadm info /dev/ttyUSB9` in our case:

```
$ udevadm info /dev/ttyUSB9
P: /devices/pci0000:00/0000:00:14.0/usb3/3-10/3-10.2/3-10.2.2/3-10.2.2.2/3-10.2.2.2:1.
↪1/ttyUSB9/tty/ttyUSB9
N: ttyUSB9
S: serial/by-id/usb-FTDI_Dual_RS232-HS-if01-port0
S: serial/by-path/pci-0000:00:14.0-usb-0:10.2.2.2:1.1-port0
E: DEVLINKS=/dev/serial/by-id/usb-FTDI_Dual_RS232-HS-if01-port0 /dev/serial/by-path/
↪pci-0000:00:14.0-usb-0:10.2.2.2:1.1-port0
E: DEVNAME=/dev/ttyUSB9
E: DEVPATH=/devices/pci0000:00/0000:00:14.0/usb3/3-10/3-10.2/3-10.2.2/3-10.2.2.2/3-10.
↪2.2.2:1.1/ttyUSB9/tty/ttyUSB9
E: ID_BUS=usb
E: ID_MODEL=Dual_RS232-HS
E: ID_MODEL_ENC=Dual\x20RS232-HS
E: ID_MODEL_FROM_DATABASE=FT2232C Dual USB-UART/FIFO IC
E: ID_MODEL_ID=6010
```

```
E: ID_PATH=pci-0000:00:14.0-usb-0:10.2.2.2:1.1
E: ID_PATH_TAG=pci-0000_00_14_0-usb-0_10_2_2_2_1_1
E: ID_REVISION=0700
E: ID_SERIAL=FTDI_Dual_RS232-HS
E: ID_TYPE=generic
E: ID_USB_DRIVER=ftdi_sio
E: ID_USB_INTERFACES=:ffffff:
E: ID_USB_INTERFACE_NUM=01
E: ID_VENDOR=FTDI
E: ID_VENDOR_ENC=FTDI
E: ID_VENDOR_FROM_DATABASE=Future Technology Devices International, Ltd
E: ID_VENDOR_ID=0403
E: MAJOR=188
E: MINOR=9
E: SUBSYSTEM=tty
E: TAGS=:systemd:
E: USEC_INITIALIZED=9129609697
```

We use the `ID_USB_INTERFACE_NUM` to distinguish between the two ports:

```
USBSerialPort:
  match:
    '@sys_name': '3-10.2.2.2'
    'ID_USB_INTERFACE_NUM': '01'
```

Matching a USB UART by Serial Number

Most of the USB serial converters in our lab have been programmed with unique serial numbers. This makes it easy to always match the same one even if the USB topology changes or a board has been moved between host systems.

```
USBSerialPort:
  match:
    'ID_SERIAL_SHORT': 'P-00-00679'
```

To check if your device has a serial number, you can use `udevadm info`:

```
$ udevadm info /dev/ttyUSB5 | grep SERIAL_SHORT
E: ID_SERIAL_SHORT=P-00-00679
```

Drivers

SerialDriver

A SerialDriver connects to a serial port. It requires one of the serial port resources.

Binds to:

- *NetworkSerialPort*
- *RawSerialPort*
- *USBSerialPort*

```
SerialDriver:
  txdelay: 0.05
```

Implements:

- *ConsoleProtocol*

Arguments:

- txdelay (float): time in seconds to wait before sending each byte

ShellDriver

A ShellDriver binds on top of a *ConsoleProtocol* and is designed to interact with a login prompt and a Linux shell.

Binds to:

- *ConsoleProtocol* (see *SerialDriver*)

Implements:

- *CommandProtocol*

```
ShellDriver:
  prompt: 'root@\w+:[^ ]+ '
  login_prompt: ' login: '
  username: 'root'
```

Arguments:

- prompt (regex): prompt to match after logging in
- login_prompt (regex): match for the login prompt
- username (str): username to use during login
- password (str): password to use during login
- keyfile (str): optional keyfile to upload after login, making the *SSHDriver* usable

SSHDriver

A SSHDriver requires a *NetworkService* resource and allows the execution of commands and file upload via network.

Binds to:

- *NetworkService*

Implements:

- *CommandProtocol*
- *FileTransferProtocol*

```
SSHDriver:
  keyfile: example.key
```

Arguments:

- keyfile (str): filename of private key to login into the remote system

InfoDriver

An InfoDriver provides an interface to retrieve system settings and state. It requires a *CommandProtocol*.

Binds to:

- `CommandProtocol` (see *ShellDriver*)

Implements:

- `InfoProtocol`

```
InfoDriver: {}
```

Arguments:

- None

UBootDriver

A UBootDriver interfaces with a u-boot boot loader via a *ConsoleProtocol*.

Binds to:

- `ConsoleProtocol` (see *SerialDriver*)

Implements:

- `CommandProtocol`

```
UBootDriver:
  prompt: 'Uboot> '
```

Arguments:

- prompt (regex): u-boot prompt to match
- password (str): optional u-boot unlock password
- init_commands (tuple): tuple of commands to execute after matching the prompt

BareboxDriver

A BareboxDriver interfaces with a barebox bootloader via a *ConsoleProtocol*.

Binds to:

- `ConsoleProtocol` (see *SerialDriver*)

Implements:

- `CommandProtocol`

```
BareboxDriver:
  prompt: 'barebox@[^:]+:[^ ]+ '
```

Arguments:

- prompt (regex): barebox prompt to match
- autoboot (regex, default="stop autoboot"): autoboot message to match
- interrupt (str, default="\n"): string to interrupt autoboot (use "\x03" for CTRL-C)

- bootstring (regex, default="Linux version d"): succesfully jumped into the kernel

ExternalConsoleDriver

An ExternalConsoleDriver implements the *ConsoleProtocol* on top of a command executed on the local computer.

Implements:

- *ConsoleProtocol*

```
ExternalConsoleDriver:
  cmd: 'microcom /dev/ttyUSB2'
  txdelay: 0.05
```

Arguments:

- cmd (str): command to execute and then bind to.
- txdelay (float): time in seconds to wait before sending each byte

AndroidFastbootDriver

An AndroidFastbootDriver allows the upload of images to a device in the USB fastboot state.

Binds to:

- *AndroidFastboot*

Implements:

- None (yet)

```
AndroidFastbootDriver:
  image: mylocal.image
```

Arguments:

- image (str): filename of the image to upload to the device

OpenOCDDriver

An OpenOCDDriver controls OpenOCD to bootstrap a target with a bootloader.

Binds to:

- *AlteraUSBBlaster*

Implements:

- *BootstrapProtocol*

Arguments:

- config (str): OpenOCD configuration file
- search (str): include search path for scripts
- image (str): filename of image to bootstrap onto the device

ManualPowerDriver

A ManualPowerDriver requires the user to control the target power states. This is required if a strategy is used with the target, but no automatic power control is available.

Implements:

- *PowerProtocol*

```
ManualPowerDriver:
  name: 'example-board'
```

Arguments:

- name (str): name of the driver (will be displayed during interaction)

ExternalPowerDriver

An ExternalPowerDriver is used to control a target power state via an external command.

Implements:

- *PowerProtocol*

```
ExternalPowerDriver:
  cmd_on: example_command on
  cmd_off: example_command off
  cmd_cycle: example_command cycle
```

Arguments:

- cmd_on (str): command to turn power to the board on
- cmd_off (str): command to turn power to the board off
- cycle (str): optional command to switch the board off and on
- delay (float): configurable delay between off and on if cycle is not set

NetworkPowerDriver

A NetworkPowerDriver controls a *NetworkPowerPort*, allowing control of the target power state without user interaction.

Binds to:

- *NetworkPowerPort*

Implements:

- *PowerProtocol*

```
NetworkPowerDriver:
  delay: 5.0
```

Arguments:

- delay (float): optional delay between off and on

DigitalOutputPowerDriver

A DigitalOutputPowerDriver can be used to control a device with external commands and a digital output port. The digital output port is used to reset the device.

Binds to:

- *DigitalOutputProtocol*

```
DigitalOutputPowerDriver:
  cmd_on: example_command on
  cmd_off: example_command off
```

Arguments:

- cmd_on (str): command to turn power to the board on
- cmd_off (str): command to turn power to the board off
- delay (float): configurable delay between off and on

MXSUSBDriver

A MXUSBDriver is used to upload an image into a device in the mxs USB loader state. This is useful to bootstrap a bootloader onto a device.

Binds to:

- *MXSUSBLoader*
- *NetworkMXSUSBLoader*

Implements:

- *BootstrapProtocol*

```
MXSUSBDriver:
  image: mybootloader.img
```

Arguments:

- image (str): The image to bootstrap onto the target

IMXUSBDriver

A IMXUSBDriver is used to upload an image into a device in the imx USB loader state. This is useful to bootstrap a bootloader onto a device.

Binds to:

- *IMXUSBLoader*
- *NetworkIMXUSBLoader*

Implements:

- *BootstrapProtocol*

```
IMXUSBDriver:
  image: mybootloader.img
```

Arguments:

- image (str): The image to bootstrap onto the target

USBStorageDriver

A USBStorageDriver allows access to a USB stick or similar device via the *USBMassStorage* resource.

Binds to:

- *USBMassStorage*

Implements:

- None (yet)

```
USBStorageDriver: {}
```

Arguments:

- None

OneWirePIODriver

A OneWirePIODriver controls a *OneWirePIO* resource. It can set and get the current state of the resource.

Binds to:

- *OneWirePIO*

Implements:

- *DigitalOutputProtocol*

```
OneWirePIODriver: {}
```

Arguments:

- None

QEMUDriver

The QEMUDriver allows the usage of a qemu instance as a target. It requires several arguments, listed below. The kernel, flash, rootfs and dtb arguments refer to images and paths declared in the environment configuration.

Binds to:

- None

```
QEMUDriver:
  qemu_bin: qemu_arm
  machine: vexpress-a9
  cpu: cortex-a9
  memory: 512M
  boot_args: "root=/dev/root console=ttyAMA0,115200"
  extra_args: ""
  kernel: kernel
  rootfs: rootfs
  dtb: dtb
```

```
tools:
  qemu_arm: /bin/qemu-system-arm
paths:
  rootfs: ../images/root
images:
  dtb: ../images/mydtb.dtb
  kernel: ../images/vmlinuz
```

Implements:

- *ConsoleProtocol*
- *PowerProtocol*

Arguments:

- qemu_bin (str): reference to the tools key for the QEMU binary
- machine (str): QEMU machine type
- cpu (str): QEMU cpu type
- memory (str): QEMU memory size (ends with M or G)
- extra_args (str): extra QEMU arguments, they are passed directly to the QEMU binary
- boot_args (str): optional, additional kernel boot argument
- kernel (str): optional, reference to the images key for the kernel
- disk (str): optional, reference to the images key for the disk image
- flash (str): optional, reference to the images key for the flash image
- rootfs (str): optional, reference to the paths key for use as the virtio-9p filesystem
- dtb (str): optional, reference to the image key for the device tree

The qemudriver also requires the specification of:

- a tool key, this contains the path to the qemu binary
- an image key, the path to the kernel image and optionally the dtb key to specify the build device tree
- a path key, this is the path to the rootfs

Strategies

Strategies are used to ensure that the device is in a certain state during a test. Such a state could be the boot loader or a booted Linux kernel with shell.

BareboxStrategy

A BareboxStrategy has three states:

- unknown
- barebox
- shell

to transition to the shell state:

```
t = get_target("main")
s = BareboxStrategy(t)
s.transition("shell")
```

this command would transition from the boot loader into a Linux shell and activate the shelldriver.

ShellStrategy

A ShellStrategy has three states:

- unknown
- off
- shell

to transition to the shell state:

```
t = get_target("main")
s = ShellStrategy(t)
s.transition("shell")
```

this command would transition directly into a Linux shell and activate the shelldriver.

UBootStrategy

A UBootStrategy has three states:

- unknown
- uboot
- shell

to transition to the shell state:

```
t = get_target("main")
s = UBootStrategy(t)
s.transition("shell")
```

this command would transition from the boot loader into a Linux shell and activate the shelldriver.

Environment Configuration

The environment configuration for a test environment consists of a YAML file which contains targets, drivers and resources. The invocation order of objects is important here since drivers may depend on other drivers or resources.

The skeleton for an environment consists of:

```
targets:
  <target-1>:
    resources:
      <resource-1>:
        <resource-1 parameters>
      <resource-2>:
        <resource-2 parameters>
    drivers:
```

```
      <driver-1>:
        <driver-1 parameters>
      <driver-2>: {} # no parameters for driver-2
  <target-2>:
    resources:
      <resources>
    drivers:
      <drivers>
  <more targets>
options:
  <option-1 name>: <value for option-1>
  <more options>
images:
  <image-1 name>: <absolute or relative path for image-1>
  <more images>
tools:
  <tool-1 name>: <absolute or relative path for tool-1>
  <more tools>
```

If you have a single target in your environment, name it "main", as the `get_target` function defaults to "main".

All the resources and drivers in this chapter have a YAML example snippet which can simply be added (at the correct indentation level, one level deeper) to the environment configuration.

Exporter Configuration

The exporter is configured by using a YAML file (with a syntax similar to the environment configs used for pytest) or by instantiating the *`Environment`* object. To configure the exporter, you need to define one or more *resource groups*, each containing one or more *resources*. This allows the exporter to group resources for various usage scenarios, e.g. all resources of a specific place or for a specific test setup. For information on how the exporter fits into the rest of labgrid, see *Remote Resources and Places*.

The basic structure of an exporter configuration file is:

```
<group-1>:
  <resources>
<group-2>:
  <resources>
```

The simplest case is with one group called "group1" containing a single *`USBSerialPort`*:

```
group1:
  USBSerialPort:
    match:
      '@sys_name': '3-1.3'
```

To reduce the amount of repeated declarations when many similar resources need to be exported, the Jinja2 template engine is used as a preprocessor for the configuration file:

```
## Iterate from group 1001 to 1016
# for idx in range(1, 17)
{{ 1000 + idx }}:
  NetworkSerialPort:
    {host: rl1, port: {{ 4000 + idx }}}
  NetworkPowerPort:
    # if 1 <= idx <= 8
```

```
    {model: apc, host: apc1, index: {{ idx }}}
    # elif 9 <= idx <= 12
    {model: netio, host: netio4, index: {{ idx - 8 }}}
    # elif 13 <= idx <= 16
    {model: netio, host: netio5, index: {{ idx - 12 }}}
    # endif
# endfor
```

Use # for line statements (like the for loops in the example) and ## for line comments. Statements like `{{ 4000 + idx }}` are expanded based on variables in the Jinja2 template.

CHAPTER 6

Development

The first step is to install labgrid into a local virtualenv.

Installation

Clone the git repository:

```
git clone https://github.com/labgrid-project/labgrid && cd labgrid
```

Create and activate a virtualenv for labgrid:

```
virtualenv -p python3 venv
source venv/bin/activate
```

Install required dependencies:

```
sudo apt install libow-dev
```

Install the development requirements:

```
pip install -r dev-requirements.txt
```

Install labgrid into the virtualenv in editable mode:

```
pip install -e .
```

Tests can now be run via:

```
python -m pytest --lg-env <config>
```

Writing a Driver

To develop a new driver for labgrid, you need to decide which protocol to implement, or implement your own protocol. If you are unsure about a new protocol's API, just use the driver directly from the client code, as deciding on a good API will be much easier when another similar driver is added.

Labgrid uses the attrs library for internal classes. First of all import attr, the protocol and the common driver class into your new driver file.

```
import attr

from labgrid.driver.common import Driver
from labgrid.protocol import ConsoleProtocol
```

Next, define your new class and list the protocols as subclasses of the new driver class. Try to avoid subclassing existing other drivers, as this limits the flexibility provided by connecting drivers and resources on a given target at runtime.

```
import attr

from labgrid.driver.common import Driver
from labgrid.protocol import ConsoleProtocol

@attr.s(cmp=False)
class ExampleDriver(Driver, ConsoleProtocol):
    pass
```

The ConsoleExpectMixin is a mixin class to add expect functionality to any class supporting the *ConsoleProtocol* and has to be the first item in the subclass list. Using the mixin class allows sharing common code, which would otherwise need to be added into multiple drivers.

```
import attr

from labgrid.driver.common import Driver
from labgrid.driver.consoleexpectmixin import ConsoleExpectMixin
from labgrid.protocol import ConsoleProtocol

@attr.s(cmp=False)
class ExampleDriver(ConsoleExpectMixin, Driver, ConsoleProtocol)
    pass
```

Additionally the driver needs to be registered with the *target_factory* and provide a bindings dictionary, so that the *Target* can resolve dependencies on other drivers or resources.

```
import attr

from labgrid.factory import target_factory
from labgrid.driver.common import Driver
from labgrid.driver.consoleexpectmixin import ConsoleExpectMixin
from labgrid.protocol import ConsoleProtocol

@target_factory.reg_driver
@attr.s(cmp=False)
class ExampleDriver(ConsoleExpectMixin, Driver, ConsoleProtocol)
    bindings = { "port": SerialPort }
    pass
```

The listed resource `SerialPort` will be bound to `self.port`, making it usable in the class. Checks are performed that the target which the driver binds to has a SerialPort, otherwise an error will be raised.

If you need to do something during instantiation, you need to add a `__attr_post_init__` method (instead of the usual `__init__` used for non-attr-classes). The minimum requirement is a call to `super().__attr_post_init__()`.

```
import attr

from labgrid.factory import target_factory
from labgrid.driver.common import Driver
from labgrid.driver.consoleexpectmixin import ConsoleExpectMixin
from labgrid.protocol import ConsoleProtocol

@target_factory.reg_driver
@attr.s(cmp=False)
class ExampleDriver(ConsoleExpectMixin, Driver, ConsoleProtocol)
    bindings = { "port": SerialPort }

    def __attr_post_init__(self):
        super().__attr_post_init__()
```

All that's left now is to implement the functionality described by the used protocol, by using the API of the bound drivers and resources.

Writing a Resource

To add a new resource to labgrid, we import attr into our new resource file. Additionally we need the *target_factory* and the common Resource class.

```
import attr

from labgrid.factory import target_factory
from labgrid.driver.common import Resource
```

Next we add our own resource with the `Resource` parent class and register it with the *target_factory*.

```
import attr

from labgrid.factory import target_factory
from labgrid.driver.common import Resource


@target_factory.reg_resource
@attr.s(cmp=False)
class ExampleResource(Resource):
    pass
```

All that is left now is to add attributes via `attr.ib()` member variables.

```
import attr

from labgrid.factory import target_factory
from labgrid.driver.common import Resource


```

```
@target_factory.reg_resource
@attr.s(cmp=False)
class ExampleResource(Resource):
    examplevar1 = attr.ib()
    examplevar2 = attr.ib()
```

The `attr.ib()` style of member definition also supports defaults and validators, see the attrs documentation.

Writing a Strategy

Labgrid only offers two basic strategies, for complex use cases a customized strategy is required. Start by creating a strategy skeleton:

```
import enum

import attr

from labgrid.step import step
from labgrid.driver.common import Strategy

class Status(enum.Enum):
    unknown = 0

class MyStrategy(Strategy):
    bindings = {
    }

    status = attr.ib(default=Status.unknown)

    @step
    def transition(self, status, *, step):
        if not isinstance(status, Status):
            status = Status[status]
        if status == Status.unknown:
            raise StrategyError("can not transition to {}".format(status))
        elif status == self.status:
            step.skip("nothing to do")
            return  # nothing to do
        else:
            raise StrategyError(
                "no transition found from {} to {}".
                format(self.status, status)
            )
        self.status = status
```

The `bindings` variable needs to declare the drivers necessary for the strategy, usually one for power, boot loader and shell. The `Status` class needs to be extended to cover the states of your strategy, then for each state an `elif` entry in the transition function needs to be added.

Lets take a look at the builtin *BareboxStrategy*. The Status enum for Barebox:

```
class Status(enum.Enum):
    unknown = 0
    barebox = 1
    shell = 2
```

defines 2 custom states and the *unknown* state as the start point. These two states are handled in the transition function:

```
elif status == Status.barebox:
    # cycle power
    self.target.activate(self.power)
    self.power.cycle()
    # interrupt barebox
    self.target.activate(self.barebox)
elif status == Status.shell:
    # tansition to barebox
    self.transition(Status.barebox)
    self.barebox.boot("")
    self.barebox.await_boot()
    self.target.activate(self.shell)
```

Here the *barebox* state simply cycles the board and activates the driver, while the *shell* state uses the barebox state to cycle the board and than boot the linux kernel.

Contributing

Thank you for thinking about contributing to labgrid! Some different backgrounds and use-cases are essential for making labgrid work well for all users.

The following should help you with submitting your changes, but don't let these guidelines keep you from opening a pull request. If in doubt, we'd prefer to see the code earlier as a work-in-progress PR and help you with the submission process.

Workflow

- Changes should be submitted via a GitHub pull request.
- Try to limit each commit to a single conceptual change.
- Add a signed-of-by line to your commits according to the *Developer's Certificate of Origin* (see below).
- Check that the tests still work before submitting the pull request. Also check the CI's feedback on the pull request after submission.
- When adding new drivers or resources, please also add the corresponding documentation and test code.
- If your change affects backward compatibility, describe the necessary changes in the commit message and update the examples where needed.

Code

- Follow the **PEP 8** style.
- Use attr.ib attributes for public attributes of your drivers and resources.
- Use isort to sort the import statements.

Documentation

- Use semantic linefeeds in .rst files.

Developer's Certificate of Origin

Labgrid uses the Developer's Certificate of Origin 1.1 with the same process as used for the Linux kernel:

> Developer's Certificate of Origin 1.1
>
> By making a contribution to this project, I certify that:
>
> 1. 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
> 2. 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
> 3. The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
> 4. 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.

Then you just add a line (using `git commit -s`) saying:

> Signed-off-by: Random J Developer <random@developer.example.org>

using your real name (sorry, no pseudonyms or anonymous contributions).

Ideas

Auto-Installer Tool

To simplify using labgrid for provisioning several boards in parallel, we should add a new tool which reads a YAML file defining several targets and a Python script to be run for each board. This tool would spawn a child process for each target, which waits until a matching resource becomes available and then executes the script.

For example, it would make it simple to load a bootloader via the *`BootstrapProtocol`*, use the *`AndroidFastbootDriver`* to upload a kernel with initramfs and then write the target's eMMC over a USB Mass Storage gadget.

Driver Priorities

In more complex use-cases, we often have multiple drivers implementing the same Protocols on the same *`Target`*. For example:

CommandProtocol (ShellDriver and SSHDriver): The SSHDriver may not be active all the time, but should be preferred when it is.

ResetProtocol (DigitalOutputResetDriver and NetworkPowerPort via power cycling): This will occour when we implement the *ResetProtocol* as below. The real reset driver should be preferred in that case.

To avoid a central precedence list (which would be problematic for third-party drivers), each driver should declare its precedence per protocol relative other drivers by referencing them by class name. This way, the Target can sort them at runtime.

Driver Preemption

To allow better handling of unexpected reboots or crashes, inactive Drivers could register callbacks on their providers (for example the BareboxDriver it's ConsoleProtocol). These callbacks would look for indications that the Target has changed state unexpectedly (by looking for the bootloader startup messages, in this case). The inactive Driver could then cause a preemption and would be activated. The current caller of the originally active driver would be notified via an exception.

File Transfer to Exporters

Currently, the exporter and client expect to have a shared filesystem (see for example how the `AndroidFastbootDriver` works when accessing a `NetworkAndroidFastboot` resource). To remove this limitation, we should have a common way to make files available to the exporter, possibly by generating a hash locally and rsyncing new files to the exporter.

Multiple Driver Instances

For some Protocols, it seems useful to allow multiple instances.

DigitalOutputProtocol: A board may have two jumpers to control the boot mode in addition to a reset GPIO. Currently it's not possible to use these on a single target.

ConsoleProtocol: Some boards have multiple console interfaces or expose a login prompt via a USB serial gadget. In most cases, it would be enough to allow switching between them.

PowerProtocol: In some cases, multiple power ports need to be controled for one Target.

Remote Target Reservation

For integration with CI systems (like Jenkins), it would help if the CI job could reserve and wait for a specific target. This could be done by managing a list of waiting users in the coordinator and notifying the current user on each invocation of labgrid-client that another user is waiting. The reservation should expire after some time if it is not used to lock the target after it becomes available.

ResetProtocol

Resetting a board is a distinct operation from cycling the power and is often triggered by pushing a button (automated via a relays or FET). If a real reset is unavailable, power cycling could be used to emulate the reset. Currently, the `DigitalOutputPowerDriver` implements the `PowerProtocol` instead, mixing the two aspects.

To handle falling back to emulation via the PowerProtocol nicely, we would need to implement *Driver Priorities*

Step Tracing

The Step infrastructure already collects timing and nesting information on executed commands, but is currently only used for in pytest or via the standalone StepReporter. By writing these events to a file (or sqlite database) as a trace, we can collect data over multiple runs for later analysis. This would become more useful by passing recognized events (stack traces, crashes, ...) and benchmark results via the Step infrastructure.

Strategy Support for labgrid-client

Currently, the client instantiates the Target, Resources and Drivers directly. By passing an environment YAML file, we could also instantiate any custom Strategy and configure image paths. This would allow us to use the Strategy to transition the board to a specific state before connecting to the console.

Target Feature Flags

It would be useful to support configuring feature flags in the target YAML definition. Then individual tests could be skipped if a required feature is unavailable on the current target without manually modifying the test suite.

CHAPTER 7

Design Decisions

This document outlines the design decisions influencing the development of labgrid.

Out of Scope

Out of scope for labgrid are:

Integrated Build System

In contrast to some other tools, labgrid explicitly has no support for building target binaries or images.

Our reasons for this are:

- Several full-featured build systems already exist and work well.
- We want to test unmodified images produced by any build system (OE/Yocto, PTXdist, Buildroot, Debian, . . .).

Test Infrastructure

Labgrid does not include a test framework.

The main reason is that with pytest we already have a test framework which:

- makes it easy to write tests
- reduces boilerplate code with flexible fixtures
- is easy to extend and has many available plugins
- allows using any Python library for creating inputs or processing outputs
- supports test report generation

Furthermore, the hardware control functionality needed for testing is also very useful during development, provisioning and other areas, so we don't want to hide that behind another test framework.

In Scope

- usable as a library for hardware provisioning
- device control via:
  - serial console
  - SSH
  - file management
  - power and reset
- emulation of external services:
  - USB stick emulation
  - external update services (Hawkbit)
- bootstrap services:
  - fastboot
  - imxusbloader

Further Goals

- tests should be equivalent for workstations and servers
- discoverability of available boards
- distributed board access

CHAPTER 8

Changes

Release 0.1.0 (released May 11, 2017)

This is the initial release of labgrid.

CHAPTER 9

Modules

labgrid package

Subpackages

labgrid.driver package

Subpackages

labgrid.driver.power package

Submodules

labgrid.driver.power.apc module

`labgrid.driver.power.apc.`**`set`**(*host*, *index*, *value*)

`labgrid.driver.power.apc.`**`get`**(*host*, *index*)

labgrid.driver.power.digipower module

`labgrid.driver.power.digipower.`**`set`**(*host*, *index*, *value*)

`labgrid.driver.power.digipower.`**`get`**(*host*, *index*)

labgrid.driver.power.gude module

`labgrid.driver.power.gude.`**`set`**(*host*, *index*, *value*)

`labgrid.driver.power.gude.`**`get`**(*host*, *index*)

labgrid.driver.power.netio module

labgrid.driver.power.netio.**set**(*host*, *index*, *value*)

labgrid.driver.power.netio.**get**(*host*, *index*)

Submodules

labgrid.driver.bareboxdriver module

class labgrid.driver.bareboxdriver.**BareboxDriver**(*target*, *prompt=''*, *autoboot='stop autoboot'*, *interrupt='n'*, *bootstring='Linux version \d'*)

Bases: *labgrid.driver.commandmixin.CommandMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.commandprotocol.CommandProtocol*, *labgrid.protocol.linuxbootprotocol.LinuxBootProtocol*

BareboxDriver - Driver to control barebox via the console. BareboxDriver binds on top of a ConsoleProtocol.

Parameters **prompt** (*str*) – The default Barebox Prompt

bindings = {'console': <class 'labgrid.protocol.consoleprotocol.ConsoleProtocol'>}

prompt = Attribute(name='prompt', default='', validator=<instance_of validator for type <class 'str'>>, repr=True, cmp

autoboot = Attribute(name='autoboot', default='stop autoboot', validator=<instance_of validator for type <class 'str'>

interrupt = Attribute(name='interrupt', default='\n', validator=<instance_of validator for type <class 'str'>>, repr=Tr

bootstring = Attribute(name='bootstring', default='Linux version \\d', validator=<instance_of validator for type <clas

on_activate()
Activate the BareboxDriver

This function checks for a prompt and awaits it if not already active

on_deactivate()
Deactivate the BareboxDriver

Simply sets the internal status to 0

run(*cmd: str*, ***, *step*)
Runs the specified command on the shell and returns the output.

Parameters **cmd** (*str*) – command to run on the shell

Returns if successful, None otherwise

Return type Tuple[List[str],List[str], int]

run_check(*cmd: str*)
Runs the specified command on the shell and returns the output if successful, raises ExecutionError otherwise.

Parameters **cmd** (*str*) – command to run on the shell

Returns stdout of the executed command

Return type List[str]

reset()
Reset the board via a CPU reset

get_status()
Retrieve status of the BareboxDriver 0 means inactive, 1 means active.

Returns status of the driver

Return type int

await_boot()
Wait for the initial Linux version string to verify we succesfully jumped into the kernel.

boot(*name: str*)
Boot the default or a specific boot entry

Parameters **name** (*str*) – name of the entry to boot

labgrid.driver.commandmixin module

class labgrid.driver.commandmixin.**CommandMixin**
Bases: object

CommandMixin implementing common functions for drivers which support the CommandProtocol

wait_for(*cmd*, *pattern*, *timeout=30.0*, *sleepduration=1*)

labgrid.driver.common module

class labgrid.driver.common.**Driver**(*target*)
Bases: *labgrid.binding.BindingMixin*

Represents a driver which is used externally or by other drivers. It implements functionality based on directly accessing the Resource or by building on top of other Drivers.

Life cycle: - create - bind (n times) - activate - usage - deactivate

labgrid.driver.common.**check_file**(*filename*, *, *command_prefix=[]*)

labgrid.driver.consoleexpectmixin module

class labgrid.driver.consoleexpectmixin.**ConsoleExpectMixin**
Bases: object

Console driver mixin to implement the read, write, expect and sendline methods. It uses the internal _read and _write methods.

The class using the ConsoleExpectMixin must provide a logger and a txdelay attribute.

read(*size=1*, *timeout=0.0*)

write(*data*)

sendline(*line*)

sendcontrol(*char*)

expect(*pattern*, *timeout=-1*)

resolve_conflicts(*client*)

labgrid.driver.exception module

exception labgrid.driver.exception.**ExecutionError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cr

exception labgrid.driver.exception.**CleanUpError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cr

labgrid.driver.externalconsoledriver module

class labgrid.driver.externalconsoledriver.**ExternalConsoleDriver**(*target*, *cmd*, *txdelay=0.0*)
Bases: *labgrid.driver.consoleexpectmixin.ConsoleExpectMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.consoleprotocol.ConsoleProtocol*

Driver implementing the ConsoleProtocol interface using a subprocess

cmd = Attribute(name='cmd', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cr

txdelay = Attribute(name='txdelay', default=0.0, validator=<instance_of validator for type <class 'float'>>, repr=True,

open()
Starts the subprocess, does nothing if it is already closed

close()
Stops the subprocess, does nothing if it is already closed

on_deactivate()

labgrid.driver.fake module

class labgrid.driver.fake.**FakeConsoleDriver**(*target*, *txdelay=0.0*)
Bases: *labgrid.driver.consoleexpectmixin.ConsoleExpectMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.consoleprotocol.ConsoleProtocol*

txdelay = Attribute(name='txdelay', default=0.0, validator=<instance_of validator for type <class 'float'>>, repr=True,

open()

close()

class labgrid.driver.fake.**FakeCommandDriver**(*target*)
Bases: *labgrid.driver.commandmixin.CommandMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.commandprotocol.CommandProtocol*

run(**args*)

run_check(**args*)

get_status()

class labgrid.driver.fake.**FakeFileTransferDriver**(*target*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.filetransferprotocol.FileTransferProtocol*

get(**args*)

put(**args*)

class labgrid.driver.fake.**FakePowerDriver**(*target*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.powerprotocol.PowerProtocol*

on(**args*)

off(**args*)

cycle(**args*)

labgrid.driver.fastbootdriver module

class labgrid.driver.fastbootdriver.**AndroidFastbootDriver**(*target*, *image=None*)
Bases: *labgrid.driver.common.Driver*

bindings = {'fastboot': {<class 'labgrid.resource.remote.NetworkAndroidFastboot'>, <class 'labgrid.resource.udev.And

image = Attribute(name='image', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, convert=

on_activate()

on_deactivate()

boot(*filename*)

flash(*partition*, *filename*)

labgrid.driver.infodriver module

class labgrid.driver.infodriver.**InfoDriver**(*target*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.infoprotocol.InfoProtocol*

InfoDriver implementing the InfoProtocol on top of CommandProtocol drivers

bindings = {'command': <class 'labgrid.protocol.commandprotocol.CommandProtocol'>}

get_ip(*interface='eth0'*)
Returns the IP of the supplied interface

get_service_status(*service*)
Returns True if service is active, False in all other cases

get_hostname()

labgrid.driver.onewiredriver module

class labgrid.driver.onewiredriver.**OneWirePIODriver**(*target*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.digitaloutputprotocol.DigitalOutputProtocol*

bindings = {'port': <class 'labgrid.resource.onewireport.OneWirePIO'>}

set(*status*)

get()

labgrid.driver.openocddriver module

class labgrid.driver.openocddriver.**OpenOCDDriver**(*target*, *config*, *search=None*, *image=None*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.bootstrapprotocol.BootstrapProtocol*

bindings = {'interface': {<class 'labgrid.resource.remote.NetworkAlteraUSBBlaster'>, <class 'labgrid.resource.udev.Al

config = Attribute(name='config', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=T

search = Attribute(name='search', default=None, validator=<optional validator for <instance_of validator for type <cla

image = Attribute(name='image', default=None, validator=<optional validator for <instance_of validator for type <class

load(*filename=None*)

labgrid.driver.powerdriver module

class labgrid.driver.powerdriver.**ManualPowerDriver**(*target*, *name*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.powerprotocol.PowerProtocol*

ManualPowerDriver - Driver to tell the user to control a target's power

name = Attribute(name='name', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True

on()

off()

cycle()

class labgrid.driver.powerdriver.**ExternalPowerDriver**(*target*, *cmd_on*, *cmd_off*, *cmd_cycle=None*, *delay=2.0*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.powerprotocol.PowerProtocol*

ExternalPowerDriver - Driver using an external command to control a target's power

cmd_on = Attribute(name='cmd_on', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=

cmd_off = Attribute(name='cmd_off', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, rep

cmd_cycle = Attribute(name='cmd_cycle', default=None, validator=<optional validator for <instance_of validator for t

delay = Attribute(name='delay', default=2.0, validator=<instance_of validator for type <class 'float'>>, repr=True, cmp

on()

off()

cycle()

class labgrid.driver.powerdriver.**NetworkPowerDriver**(*target*, *delay=2.0*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.powerprotocol.PowerProtocol*

NetworkPowerDriver - Driver using a networked power switch to control a target's power

bindings = {'port': <class 'labgrid.resource.power.NetworkPowerPort'>}

delay = Attribute(name='delay', default=2.0, validator=<instance_of validator for type <class 'float'>>, repr=True, cmp

on()

off()

cycle()

get()

class labgrid.driver.powerdriver.**DigitalOutputPowerDriver**(*target*, *cmd_on*, *cmd_off*, *delay=1.0*)

Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.powerprotocol.PowerProtocol*

DigitalOutputPowerDriver - Driver using a DigitalOutput to reset the target and subprocesses to turn it on and off

bindings = {'output': <class 'labgrid.protocol.digitaloutputprotocol.DigitalOutputProtocol'>}

cmd_on = Attribute(name='cmd_on', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=

cmd_off = Attribute(name='cmd_off', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, rep

delay = Attribute(name='delay', default=1.0, validator=<instance_of validator for type <class 'float'>>, repr=True, cmp

on()

off()

cycle()

get()

labgrid.driver.qemudriver module

The QEMUDriver implements a driver to use a QEMU target

class labgrid.driver.qemudriver.**QEMUDriver**(*target*, *qemu_bin*, *machine*, *cpu*, *memory*, *extra_args*, *boot_args=None*, *kernel=None*, *disk=None*, *rootfs=None*, *dtb=None*, *flash=None*)

Bases: *labgrid.driver.consoleexpectmixin.ConsoleExpectMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.powerprotocol.PowerProtocol*, *labgrid.protocol.consoleprotocol.ConsoleProtocol*

The QEMUDriver implements an interface to start targets as qemu instances.

The kernel, flash, rootfs and dtb arguments refer to images and paths declared in the environment configuration.

Parameters

- **qemu_bin** (*str*) – reference to the tools key for the QEMU binary
- **machine** (*str*) – QEMU machine type
- **cpu** (*str*) – QEMU cpu type
- **memory** (*str*) – QEMU memory size (ends with M or G)
- **extra_args** (*str*) – extra QEMU arguments, they are passed directly to the QEMU binary
- **boot_args** (*str*) – optional, additional kernel boot argument
- **kernel** (*str*) – optional, reference to the images key for the kernel
- **disk** (*str*) – optional, reference to the images key for the disk image
- **flash** (*str*) – optional, reference to the images key for the flash image

- **rootfs** (*str*) – optional, reference to the paths key for use as the virtio-9p filesystem
- **dtb** (*str*) – optional, reference to the image key for the device tree

qemu_bin = Attribute(name='qemu_bin', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, r

machine = Attribute(name='machine', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, rep

cpu = Attribute(name='cpu', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

memory = Attribute(name='memory', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=

extra_args = Attribute(name='extra_args', default=NOTHING, validator=<instance_of validator for type <class 'str'>

boot_args = Attribute(name='boot_args', default=None, validator=<optional validator for <instance_of validator for ty

kernel = Attribute(name='kernel', default=None, validator=<optional validator for <instance_of validator for type <clas

disk = Attribute(name='disk', default=None, validator=<optional validator for <instance_of validator for type <class 'st

rootfs = Attribute(name='rootfs', default=None, validator=<optional validator for <instance_of validator for type <clas

dtb = Attribute(name='dtb', default=None, validator=<optional validator for <instance_of validator for type <class 'str'>

flash = Attribute(name='flash', default=None, validator=<optional validator for <instance_of validator for type <class '

on_activate()

on_deactivate()

on()
Start the QEMU subprocess, accept the unix socket connection and afterwards start the emulator using a QMP Command

off()
Stop the emulator using a monitor command and await the exitcode

cycle()
Cycle the emulator by restarting it

monitor_command(*command*)
Execute a monitor_command via the QMP

labgrid.driver.serialdriver module

class labgrid.driver.serialdriver.**SerialDriver**(*target*, *txdelay=0.0*)
Bases: *labgrid.driver.consoleexpectmixin.ConsoleExpectMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.consoleprotocol.ConsoleProtocol*

Driver implementing the ConsoleProtocol interface over a SerialPort connection

bindings = {'port': {<class 'labgrid.resource.base.SerialPort'>, <class 'labgrid.resource.serialport.NetworkSerialPort'>

txdelay = Attribute(name='txdelay', default=0.0, validator=<instance_of validator for type <class 'float'>>, repr=True,

on_activate()

on_deactivate()

open()
Opens the serialport, does nothing if it is already closed

close()
Closes the serialport, does nothing if it is already closed

labgrid.driver.shelldriver module

The ShellDriver provides the CommandProtocol, ConsoleProtocol and InfoProtocol on top of a SerialPort.

class labgrid.driver.shelldriver.**ShellDriver**(*target*, *prompt*, *login_prompt*, *username*, *password=''*, *keyfile=''*)

Bases: *labgrid.driver.commandmixin.CommandMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.commandprotocol.CommandProtocol*, *labgrid.protocol.filetransferprotocol.FileTransferProtocol*

ShellDriver - Driver to execute commands on the shell ShellDriver binds on top of a ConsoleProtocol.

Parameters

- **prompt** (*regex*) – The Linux Prompt to detect
- **login_prompt** (*regex*) – The Login Prompt to detect
- **username** (*str*) – username to login with
- **password** (*str*) – password to login with
- **keyfile** (*str*) – keyfile to bind mount over users authorized keys

bindings = {'console': <class 'labgrid.protocol.consoleprotocol.ConsoleProtocol'>}

prompt = Attribute(name='prompt', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=

login_prompt = Attribute(name='login_prompt', default=NOTHING, validator=<instance_of validator for type <class

username = Attribute(name='username', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, r

password = Attribute(name='password', default='', validator=<instance_of validator for type <class 'str'>>, repr=True,

keyfile = Attribute(name='keyfile', default='', validator=<instance_of validator for type <class 'str'>>, repr=True, cmp

on_activate()

on_deactivate()

run(*cmd*, *timeout=30.0*)

run_check(*cmd*, *timeout=30*)

Runs the specified cmd on the shell and returns the output if successful, raises ExecutionError otherwise.

Arguments: cmd - cmd to run on the shell

get_status()

Returns the status of the shell-driver. 0 means not connected/found, 1 means shell

put_ssh_key(*key*)

put_bytes(*buf: bytes*, *remotefile: str*)

Upload a file to the target. Will silently overwrite the remote file if it already exists.

Parameters

- **buf** (*bytes*) – file contents
- **remotefile** (*str*) – destination filename on the target

Raises

- IOError – if the provided localfile could not be found
- ExecutionError – if something else went wrong

put (*localfile: str*, *remotefile: str*)
Upload a file to the target. Will silently overwrite the remote file if it already exists.

Parameters

- **localfile** (*str*) – source filename on the local machine
- **remotefile** (*str*) – destination filename on the target

Raises

- IOError – if the provided localfile could not be found
- ExecutionError – if something else went wrong

get_bytes (*remotefile: str*)
Download a file from the target.

Parameters **remotefile** (*str*) – source filename on the target

Returns (bytes) file contents

Raises

- IOError – if localfile could be written
- ExecutionError – if something went wrong

get (*remotefile: str*, *localfile: str*)
Download a file from the target. Will silently overwrite the local file if it already exists.

Parameters

- **remotefile** (*str*) – source filename on the target
- **localfile** (*str*) – destination filename on the local machine (can be relative)

Raises

- IOError – if localfile could be written
- ExecutionError – if something went wrong

run_script (*data: bytes*, *timeout: int = 60*)
Upload a script to the target and run it.

Parameters

- **data** (*bytes*) – script data
- **timeout** (*int*) – timeout for the script to finish execution

Returns str, stderr: str, return_value: int)

Return type Tuple of (stdout

Raises

- IOError – if the provided localfile could not be found
- ExecutionError – if something else went wrong

run_script_file (*scriptfile: str*, **args*, *timeout: int = 60*)
Upload a script file to the target and run it.

Parameters

- **scriptfile** (*str*) – source file on the local file system to upload to the target
- ***args** – (list of str): any arguments for the script as positional arguments

- **timeout** (*int*) – timeout for the script to finish execution

Returns str, stderr: str, return_value: int)

Return type Tuple of (stdout

Raises

- ExecutionError – if something went wrong
- IOError – if the provided localfile could not be found

labgrid.driver.sshdriver module

The SSHDriver uses SSH as a transport to implement CommandProtocol and FileTransferProtocol

class labgrid.driver.sshdriver.**SSHDriver**(*target*, *keyfile=''*)

Bases: *labgrid.driver.commandmixin.CommandMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.commandprotocol.CommandProtocol*, *labgrid.protocol.filetransferprotocol.FileTransferProtocol*

SSHDriver - Driver to execute commands via SSH

bindings = {'networkservice': <class 'labgrid.resource.networkservice.NetworkService'>}

keyfile = Attribute(name='keyfile', default='', validator=<instance_of validator for type <class 'str'>>, repr=True, cmp

on_activate()

on_deactivate()

run(*cmd*)

Execute *cmd* on the target.

This method runs the specified *cmd* as a command on its target. It uses the ssh shell command to run the command and parses the exitcode. cmd - command to be run on the target

returns: (stdout, stderr, returncode)

run_check(*cmd*)

Runs the specified cmd on the shell and returns the output if successful, raises ExecutionError otherwise.

Arguments: cmd - cmd to run on the shell

get_status()

The SSHDriver is always connected, return 1

put(*filename*, *remotepath=None*)

get(*filename*, *destination='.'*)

labgrid.driver.ubootdriver module

The U-Boot Module contains the UBootDriver

class labgrid.driver.ubootdriver.**UBootDriver**(*target*, *prompt=''*, *password=''*, *init_commands=NOTHING*)

Bases: *labgrid.driver.commandmixin.CommandMixin*, *labgrid.driver.common.Driver*, *labgrid.protocol.commandprotocol.CommandProtocol*, *labgrid.protocol.linuxbootprotocol.LinuxBootProtocol*

UBootDriver - Driver to control uboot via the console. UBootDriver binds on top of a ConsoleProtocol.

Parameters

- **prompt** (*str*) – The default UBoot Prompt
- **password** (*str*) – optional password to unlock UBoot
- **init_commands** (*Tuple[str]*) – a tuple of commands to run after unlock

bindings = {'console': <class 'labgrid.protocol.consoleprotocol.ConsoleProtocol'>}

prompt = Attribute(name='prompt', default='', validator=<instance_of validator for type <class 'str'>>, repr=True, cmp

password = Attribute(name='password', default='', validator=<instance_of validator for type <class 'str'>>, repr=True,

init_commands = Attribute(name='init_commands', default=Factory(factory=<class 'tuple'>, takes_self=False), valida

on_activate()
Activate the UBootDriver

This function checks for a prompt and awaits it if not already active

on_deactivate()
Deactivate the UBootDriver

Simply sets the internal status to 0

run(*cmd*)
Runs the specified command on the shell and returns the output.

Parameters **cmd** (*str*) – command to run on the shell

Returns if successful, None otherwise

Return type Tuple[List[str],List[str], int]

run_check(*cmd*)
Runs the specified command on the shell and returns the output if successful, raises ExecutionError otherwise.

Parameters **cmd** (*str*) – command to run on the shell

Returns stdout of the executed command

Return type List[str]

get_status()
Retrieve status of the UBootDriver. 0 means inactive, 1 means active.

Returns status of the driver

Return type int

reset()
Reset the board via a CPU reset

await_boot()
Wait for the initial Linux version string to verify we succesfully jumped into the kernel.

boot(*name*)
Boot the default or a specific boot entry

Parameters **name** (*str*) – name of the entry to boot

labgrid.driver.usbloader module

class labgrid.driver.usbloader.**MXSUSBDriver**(*target*, *image=None*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.bootstrapprotocol.BootstrapProtocol*

bindings = {'loader': {<class 'labgrid.resource.remote.NetworkMXSUSBLoader'>, <class 'labgrid.resource.udev.MXSU

image = Attribute(name='image', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, convert=

on_activate()

on_deactivate()

load(*filename=None*)

class labgrid.driver.usbloader.**IMXUSBDriver**(*target*, *image=None*)
Bases: *labgrid.driver.common.Driver*, *labgrid.protocol.bootstrapprotocol.BootstrapProtocol*

bindings = {'loader': {<class 'labgrid.resource.remote.NetworkIMXUSBLoader'>, <class 'labgrid.resource.udev.IMXU

image = Attribute(name='image', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, convert=

on_activate()

on_deactivate()

load(*filename=None*)

labgrid.driver.usbstorage module

class labgrid.driver.usbstorage.**USBStorageDriver**(*target*)
Bases: *labgrid.driver.common.Driver*

bindings = {'storage': <class 'labgrid.resource.udev.USBMassStorage'>}

on_activate()

on_deactivate()

write_image(*filename*)

get_size()

labgrid.external package

Submodules

labgrid.external.hawkbit module

class labgrid.external.hawkbit.**HawkbitTestClient**(*host*, *port*, *username*, *password*, *version=1.0*)
Bases: object

host = Attribute(name='host', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

port = Attribute(name='port', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

username = Attribute(name='username', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, r

password = Attribute(name='password', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, r

version = Attribute(name='version', default=1.0, validator=<instance_of validator for type <class 'float'>>, repr=True,

add_target(*device_address: str*, *token: str*)
Add a target to the HawkBit Installation

Parameters

- **device_address** (–) – the device address of the added target
- **token** (–) – token to uniquely identify the target

add_swmodule()

add_distributionset()

add_artifact(*filename: str*)

assign_target()

post_json(*endpoint: str*, *data: dict*)

post_binary(*endpoint: str*, *filename: str*)

get_endpoint(*endpoint: str*)

exception labgrid.external.hawkbit.**HawkbiError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, convert

labgrid.external.usbstick module

The USBStick module provides support to interactively use a simulated USB device in a test.

class labgrid.external.usbstick.**USBStatus**
Bases: enum.Enum

This class describes the USBStick Status

unplugged = 0

plugged = 1

mounted = 2

class labgrid.external.usbstick.**USBStick**(*target*, *image_dir*, *image_name=''*)
Bases: object

The USBStick class provides an easy to use interface to describe a target as an USB Stick.

target = Attribute(name='target', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, c

image_dir = Attribute(name='image_dir', default=NOTHING, validator=<instance_of validator for type <class 'str'>>

image_name = Attribute(name='image_name', default='', validator=<instance_of validator for type <class 'str'>>, repr=

plug_in()
Insert the USBStick

This function plugs the virtual USB Stick in, making it available to the connected computer.

plug_out()
Plugs out the USBStick

Plugs out the USBStick from the connected computer, does nothing if it is already unplugged

put_file(*filename*, *destination=''*)
Put a file onto the USBStick Image

Puts a file onto the USB Stick, raises a StateError if it is not mounted on the host computer.

get_file(*filename*)
Gets a file from the USBStick Image

Gets a file from the USB Stick, raises a StateError if it is not mounted on the host computer.

upload_image(*image*)
Upload a complete image as a new USB Stick

This replaces the current USB Stick image, storing it permanently on the RiotBoard.

switch_image(*image_name*)
Switch between already uploaded images on the target.

exception labgrid.external.usbstick.**StateError**(*msg*)
Bases: Exception

Exception which indicates a error in the state handling of the test

msg = Attribute(name='msg', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, convert

labgrid.protocol package

Submodules

labgrid.protocol.bootstrapprotocol module

class labgrid.protocol.bootstrapprotocol.**BootstrapProtocol**
Bases: abc.ABC

load(*filename: str*)

labgrid.protocol.commandprotocol module

class labgrid.protocol.commandprotocol.**CommandProtocol**
Bases: abc.ABC

Abstract class for the CommandProtocol

run(*command: str*)
Run a command

run_check(*command: str*)
Run a command, return str if succesful, ExecutionError otherwise

get_status()
Get status of the Driver

wait_for()
Wait for a shell command to return with the specified output

labgrid.protocol.consoleprotocol module

class labgrid.protocol.consoleprotocol.**ConsoleProtocol**
Bases: abc.ABC

Abstract class for the ConsoleProtocol

read()
Read data from underlying port

write(*data: bytes*)
Write data to underlying port

sendline(*line: str*)

sendcontrol(*char: str*)

expect(*pattern: str*)

class Client
Bases: abc.ABC

get_console_matches()

notify_console_match(*pattern*, *match*)

labgrid.protocol.digitaloutputprotocol module

class labgrid.protocol.digitaloutputprotocol.**DigitalOutputProtocol**
Bases: abc.ABC

Abstract class providing the OneWireProtocol interface

get()
Implementations should return the status of the OneWirePort.

set(*status*)
Implementations should set the status of the OneWirePort

labgrid.protocol.filesystemprotocol module

class labgrid.protocol.filesystemprotocol.**FileSystemProtocol**
Bases: abc.ABC

read(*filename: str*)

write(*filename: str*, *data: bytes*, *append: bool*)

labgrid.protocol.filetransferprotocol module

class labgrid.protocol.filetransferprotocol.**FileTransferProtocol**
Bases: abc.ABC

put(*filename: str*, *remotepath: str*)

get(*filename: str*, *destination: str*)

labgrid.protocol.infoprotocol module

class labgrid.protocol.infoprotocol.**InfoProtocol**
Bases: abc.ABC

Abstract class providing the InfoProtocol interface

get_ip(*interface: str = 'eth0'*)
Implementations should return the IP-adress for the supplied interface.

get_hostname()
Implementations should return the hostname for the supplied interface.

get_service_status(*service*)
Implementations should return the status of a service

labgrid.protocol.linuxbootprotocol module

class labgrid.protocol.linuxbootprotocol.**LinuxBootProtocol**
Bases: abc.ABC

boot(*name: str*)

await_boot()

reset()

labgrid.protocol.mmioprotocol module

class labgrid.protocol.mmioprotocol.**MMIOProtocol**
Bases: abc.ABC

read(*address: int*, *size: int*, *count: int*) → bytes

write(*address: int*, *size: int*, *data: bytes*) → None

labgrid.protocol.powerprotocol module

class labgrid.protocol.powerprotocol.**PowerProtocol**
Bases: abc.ABC

on()

off()

cycle()

labgrid.provider package

Submodules

labgrid.provider.fileprovider module

class labgrid.provider.fileprovider.**FileProvider**
Bases: abc.ABC

Abstract class for the FileProvider

get(*name: str*) → dict
Get a dictionary of target paths to local paths for a given name.

list()
Get a list of names.

labgrid.provider.mediafileprovider module

class labgrid.provider.mediafileprovider.**MediaFileProvider**(*groups={}*)
Bases: *labgrid.provider.fileprovider.FileProvider*

groups = Attribute(name='groups', default={}, validator=<instance_of validator for type <class 'dict'>>, repr=True, cm

get(*name*)

list()

labgrid.pytestplugin package

Submodules

labgrid.pytestplugin.fixtures module

labgrid.pytestplugin.fixtures.**pytest_addoption**(*parser*)

labgrid.pytestplugin.fixtures.**env**(*request*)
Return the environment configured in the supplied configuration file. It contains the targets contained in the configuration file.

labgrid.pytestplugin.fixtures.**target**(*env*)
Return the default target *main* configured in the supplied configuration file.

labgrid.pytestplugin.reporter module

labgrid.pytestplugin.reporter.**bold**(*text*)

labgrid.pytestplugin.reporter.**under**(*text*)

labgrid.pytestplugin.reporter.**pytest_configure**(*config*)

class labgrid.pytestplugin.reporter.**StepReporter**(*terminalreporter*, *, *rewrite=False*)
Bases: object

notify(*event*)

pytest_runtest_logstart()

pytest_runtest_logreport(*report*)

labgrid.remote package

Submodules

labgrid.remote.authenticator module

class labgrid.remote.authenticator.**AuthenticatorSession**
Bases: sphinx.ext.autodoc.ApplicationSession

onJoin(*details*)

labgrid.remote.client module

The remote.client module contains the functionality to connect to a coordinator, acquire a place and interact with the connected resources

exception labgrid.remote.client.**Error**
Bases: Exception

exception labgrid.remote.client.**UserError**
Bases: *labgrid.remote.client.Error*

exception labgrid.remote.client.**ServerError**
Bases: *labgrid.remote.client.Error*

class labgrid.remote.client.**ClientSession**
Bases: sphinx.ext.autodoc.ApplicationSession

The ClientSession encapsulates all the actions a Client can Invoke on the coordinator.

onConnect()
Actions which are executed if a connection is successfully opened.

onChallenge(*challenge*)

onJoin(*details*)

on_resource_changed(*exporter*, *group_name*, *resource_name*, *resource*)

on_place_changed(*name*, *config*)

monitor()

complete()

print_resources()
Print out the resources

print_places()
Print out the places

print_who()
Print acquired places by user

get_place(*place=None*)

get_idle_place(*place=None*)

get_acquired_place(*place=None*)

print_place()
Print out the current place and related resources

add_place()
Add a place to the coordinator

del_place()
Delete a place from the coordinator

add_alias()
Add an alias for a place on the coordinator

del_alias()
Delete an alias for a place from the coordinator

set_comment()
Set the comment on a place

add_match()
Add a match for a place, making fuzzy matching available to the client

del_match()
Delete a match for a place

acquire()
Acquire a place, marking it unavailable for other clients

release()
Release a previously acquired place

get_target_resources(*place*)

get_target_config(*place*)

env()

power()

digital_io()

console()

fastboot()

bootstrap()

labgrid.remote.client.**start_session**(*url*, *realm*, *extra*)

labgrid.remote.client.**main**()

labgrid.remote.common module

class labgrid.remote.common.**ResourceEntry**(*data*, *acquired=None*)
Bases: object

data = Attribute(name='data', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, conve

acquired = Attribute(name='acquired', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, c

avail

cls

params

args
arguments for resource construction

extra
extra resource information

asdict()

class labgrid.remote.common.**ResourceMatch**(*exporter*, *group*, *cls*, *name=None*)
Bases: object

exporter = Attribute(name='exporter', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=T

group = Attribute(name='group', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, co

cls = Attribute(name='cls', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, convert=

name = Attribute(name='name', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, convert=N

classmethod fromstr(*pattern*)

ismatch(*resource_path*)

class labgrid.remote.common.**Place**(*name*, *aliases=NOTHING*, *comment=''*, *matches=NOTHING*, *acquired=None*, *acquired_resources=NOTHING*, *created=NOTHING*, *changed=NOTHING*)
Bases: object

name = Attribute(name='name', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, conv

aliases = Attribute(name='aliases', default=Factory(factory=<class 'set'>, takes_self=False), validator=None, repr=Tru

comment = Attribute(name='comment', default='', validator=None, repr=True, cmp=True, hash=None, init=True, conve

matches = Attribute(name='matches', default=Factory(factory=<class 'list'>, takes_self=False), validator=None, repr=T

acquired = Attribute(name='acquired', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, c

acquired_resources = Attribute(name='acquired_resources', default=Factory(factory=<class 'list'>, takes_self=Fals

created = Attribute(name='created', default=Factory(factory=<function Place.<lambda>>, takes_self=False), validator=

changed = Attribute(name='changed', default=Factory(factory=<function Place.<lambda>>, takes_self=False), validato

asdict()

show(*level=0*)

hasmatch(*resource_path*)
Return True if this place as a ResourceMatch object for the given resource path.

A resource_path has the structure (exporter, group, cls, name).

touch()

labgrid.remote.common.**enable_tcp_nodelay**(*session*)
asyncio/autobahn does not set TCP_NODELAY by default, so we need to do it like this for now.

labgrid.remote.config module

class labgrid.remote.config.**ResourceConfig**(*filename*)
Bases: object

filename = Attribute(name='filename', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, re

labgrid.remote.coordinator module

The coordinator module coordinates exported resources and clients accessing them.

class labgrid.remote.coordinator.**Action**
Bases: enum.Enum

An enumeration.

ADD = 0

DEL = 1

UPD = 2

class labgrid.remote.coordinator.**RemoteSession**
Bases: object

class encapsulating a session, used by ExporterSession and ClientSession

coordinator = Attribute(name='coordinator', default=NOTHING, validator=None, repr=True, cmp=True, hash=None

session = Attribute(name='session', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True

authid = Attribute(name='authid', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True,

key
Key of the session

name
Name of the session

class labgrid.remote.coordinator.**ExporterSession**(*coordinator*, *session*, *authid*)
Bases: *labgrid.remote.coordinator.RemoteSession*

An ExporterSession is opened for each Exporter connecting to the coordinator, allowing the Exporter to get and set resources

groups = Attribute(name='groups', default=Factory(factory=<class 'dict'>, takes_self=False), validator=None, repr=Tru

set_resource(*groupname*, *resourcename*, *resource*)

get_resources()
Method invoked by the exporter, get a resource from the coordinator

class labgrid.remote.coordinator.**ClientSession**(*coordinator*, *session*, *authid*)
Bases: *labgrid.remote.coordinator.RemoteSession*

acquired = Attribute(name='acquired', default=Factory(factory=<class 'list'>, takes_self=False), validator=None, repr=

class labgrid.remote.coordinator.**CoordinatorComponent**
Bases: sphinx.ext.autodoc.ApplicationSession

onConnect()

onChallenge(*challenge*)

onJoin(*details*)

save()

load()

on_session_join(*session_details*)

on_session_leave(*session_id*)

attach(*name*, *details=None*)

set_resource(*groupname*, *resourcename*, *resource*, *details=None*)

get_resources(*details=None*)

add_place(*name*, *details=None*)

del_place(*name*, *details=None*)

add_place_alias(*placename*, *alias*, *details=None*)

del_place_alias(*placename*, *alias*, *details=None*)

set_place_comment(*placename*, *comment*, *details=None*)

add_place_match(*placename*, *pattern*, *details=None*)

del_place_match(*placename*, *pattern*, *details=None*)

acquire_place(*name*, *details=None*)

release_place(*name*, *details=None*)

get_places(*details=None*)

labgrid.remote.exporter module

The remote.exporter module exports resources to the coordinator and makes them available to other clients on the same coordinator

labgrid.remote.exporter.**get_free_port**()
Helper function to always return an unused port.

class labgrid.remote.exporter.**ResourceExport**(*data*, *acquired=None*)
Bases: *labgrid.remote.common.ResourceEntry*

Represents a local resource exported via a specific protocol.

The ResourceEntry attributes contain the information for the client.

local = Attribute(name='local', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=False, co

local_params = Attribute(name='local_params', default=NOTHING, validator=None, repr=True, cmp=True, hash=N

poll()

class labgrid.remote.exporter.**USBSerialPortExport**(*data*, *acquired=None*)
Bases: *labgrid.remote.exporter.ResourceExport*

ResourceExport for a USB SerialPort

class labgrid.remote.exporter.**USBEthernetExport**(*data*, *acquired=None*)
Bases: *labgrid.remote.exporter.ResourceExport*

ResourceExport for a USB ethernet interface

class labgrid.remote.exporter.**USBGenericExport**(*data*, *acquired=None*)
Bases: *labgrid.remote.exporter.ResourceExport*

ResourceExport for USB devices accessed directly from userspace

class labgrid.remote.exporter.**ExporterSession**
Bases: sphinx.ext.autodoc.ApplicationSession

onConnect()
Set up internal datastructures on successful connection: - Setup loop, name, authid and address - Join the coordinator as an exporter

onChallenge(*challenge*)
Function invoked on received challege, returns just a dummy ticket at the moment, authentication is not supported yet

onJoin(*details*)
On successful join: - export available resources - bail out if we are unsuccessful

onLeave(*details*)
Cleanup after leaving the coordinator connection

onDisconnect()

acquire(*group_name*, *resource_name*)

release(*group_name*, *resource_name*)

poll()

add_resource(*group_name*, *resource_name*, *cls*, *params*)
Add a resource to the exporter and update status on the coordinator

update_resource(*group_name*, *resource_name*)
Update status on the coordinator

labgrid.remote.exporter.**main**()

labgrid.resource package

Submodules

labgrid.resource.base module

class labgrid.resource.base.**SerialPort**(*target*, *port=None*, *speed=115200*)
Bases: *labgrid.resource.common.Resource*

The basic SerialPort describes port and speed

Parameters

- **port** (*str*) – port to connect to
- **speed** (*int*) – speed of the port, defaults to 115200

port = Attribute(name='port', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, convert=No

speed = Attribute(name='speed', default=115200, validator=<instance_of validator for type <class 'int'>>, repr=True, cm

class labgrid.resource.base.**EthernetInterface**(*target*, *ifname=None*)
Bases: *labgrid.resource.common.Resource*

The basic EthernetInterface contains an interfacename

Parameters **ifname** (*str*) – name of the interface

ifname = Attribute(name='ifname', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, conve

labgrid.resource.common module

class labgrid.resource.common.**Resource**(*target*)
Bases: *labgrid.binding.BindingMixin*

Represents a resource which is used by drivers. It only stores information and does not implement any actual functionality.

Resources can exist without a target, but they must be bound to one before use.

Life cycle:

•create

•bind (n times)

avail = Attribute(name='avail', default=True, validator=<instance_of validator for type <class 'bool'>>, repr=True, cm

command_prefix

class labgrid.resource.common.**NetworkResource**(*target*, *host*)
Bases: *labgrid.resource.common.Resource*

Represents a remote Resource available on another computer.

This stores a command_prefix to describe how to connect to the remote computer.

Parameters **host** (*str*) – remote host the resource is available on

host = Attribute(name='host', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

command_prefix

class labgrid.resource.common.**ResourceManager**
Bases: object

instances = {}

classmethod get()

on_resource_added(*resource*)

poll()

class labgrid.resource.common.**ManagedResource**(*target*)
Bases: *labgrid.resource.common.Resource*

Represents a resource which can appear and disappear at runtime. Every ManagedResource has a corresponding ResourceManager which handles these events.

manager_cls
alias of *ResourceManager*

timeout = Attribute(name='timeout', default=2.0, validator=<instance_of validator for type <class 'float'>>, repr=True,

poll()

labgrid.resource.networkservice module

class labgrid.resource.networkservice.**NetworkService**(*target*, *address*, *username*)
Bases: *labgrid.resource.common.Resource*

address = Attribute(name='address', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr

username = Attribute(name='username', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, r

labgrid.resource.onewireport module

class labgrid.resource.onewireport.**OneWirePIO**(*target*, *host*, *path*)
Bases: *labgrid.resource.common.Resource*

This resource describes a Onewire PIO Port.

Parameters

- **host** (*str*) – hostname of the owserver e.g. localhost:4304
- **path** (*str*) – path to the port on the owserver e.g. 29.7D6913000000/PIO.0

host = Attribute(name='host', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

path = Attribute(name='path', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

labgrid.resource.power module

class labgrid.resource.power.**NetworkPowerPort**(*target*, *model*, *host*, *index*)
Bases: *labgrid.resource.common.Resource*

The NetworkPowerPort describes a remotely switchable PowerPort

Parameters

- **model** (*str*) – model of the external power switch
- **host** (*str*) – host to connect to
- **index** (*str*) – index of the power port on the external switch

model = Attribute(name='model', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=Tru

host = Attribute(name='host', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

index = Attribute(name='index', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=Tru

labgrid.resource.remote module

class labgrid.resource.remote.**RemotePlaceManager**
Bases: *labgrid.resource.common.ResourceManager*

on_resource_added(*resource*)

poll()

class labgrid.resource.remote.**RemotePlace**(*target*, *name*)
Bases: *labgrid.resource.common.ManagedResource*

manager_cls
alias of *RemotePlaceManager*

name = Attribute(name='name', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

class labgrid.resource.remote.**RemoteUSBResource**(*target*, *host*, *busnum*, *devnum*, *path*, *vendor_id*, *model_id*)
Bases: *labgrid.resource.common.NetworkResource*, *labgrid.resource.common.ManagedResource*

manager_cls
alias of *RemotePlaceManager*

busnum = Attribute(name='busnum', default=NOTHING, validator=<optional validator for <instance_of validator for ty

devnum = Attribute(name='devnum', default=NOTHING, validator=<optional validator for <instance_of validator for ty

path = Attribute(name='path', default=NOTHING, validator=<optional validator for <instance_of validator for type <cl

vendor_id = Attribute(name='vendor_id', default=NOTHING, validator=<optional validator for <instance_of validator

model_id = Attribute(name='model_id', default=NOTHING, validator=<optional validator for <instance_of validator fo

class labgrid.resource.remote.**NetworkAndroidFastboot**(*target*, *host*, *busnum*, *devnum*, *path*, *vendor_id*, *model_id*)
Bases: *labgrid.resource.remote.RemoteUSBResource*

class labgrid.resource.remote.**NetworkIMXUSBLoader**(*target*, *host*, *busnum*, *devnum*, *path*, *vendor_id*, *model_id*)
Bases: *labgrid.resource.remote.RemoteUSBResource*

class labgrid.resource.remote.**NetworkMXSUSBLoader**(*target*, *host*, *busnum*, *devnum*, *path*, *vendor_id*, *model_id*)
Bases: *labgrid.resource.remote.RemoteUSBResource*

class labgrid.resource.remote.**NetworkAlteraUSBBlaster**(*target*, *host*, *busnum*, *devnum*, *path*, *vendor_id*, *model_id*)
Bases: *labgrid.resource.remote.RemoteUSBResource*

labgrid.resource.serialport module

class labgrid.resource.serialport.**RawSerialPort**(*target*, *port=None*, *speed=115200*)
Bases: *labgrid.resource.base.SerialPort*, *labgrid.resource.common.Resource*

RawSerialPort describes a serialport which is vailable on the local computer.

class labgrid.resource.serialport.**NetworkSerialPort**(*target*, *host*, *port*, *speed=115200*)
Bases: *labgrid.resource.common.NetworkResource*

A NetworkSerialPort is a remotely accessable serialport, usually accessed via rfc2217.

Parameters

- **port** (*str*) – connection string to the port e.g. 'rfc2217://<host>:<port>'
- **speed** (*int*) – speed of the port e.g. 9800

port = Attribute(name='port', default=NOTHING, validator=<optional validator for <instance_of validator for type <cl

speed = Attribute(name='speed', default=115200, validator=<instance_of validator for type <class 'int'>>, repr=True, cr

labgrid.resource.udev module

class labgrid.resource.udev.**UdevManager**
Bases: *labgrid.resource.common.ResourceManager*

on_resource_added(*resource*)

poll()

class labgrid.resource.udev.**USBResource**(*target*, *match*, *device=None*)
Bases: *labgrid.resource.common.ManagedResource*

manager_cls
alias of *UdevManager*

match = Attribute(name='match', default=NOTHING, validator=<instance_of validator for type <class 'dict'>>, repr=T

device = Attribute(name='device', default=None, validator=None, repr=True, cmp=True, hash=False, init=True, conver

filter_match(*device*)

try_match(*device*)

update()

busnum

devnum

path

vendor_id

model_id

read_attr(*attribute*)
read uncached attribute value from sysfs

pyudev currently supports only cached access to attributes, so we read directly from sysfs.

class labgrid.resource.udev.**USBSerialPort**(*target*, *match*, *device=None*, *port=None*, *speed=115200*)
Bases: *labgrid.resource.base.SerialPort*, *labgrid.resource.udev.USBResource*

update()

class labgrid.resource.udev.**USBMassStorage**(*target*, *match*, *device=None*)
Bases: *labgrid.resource.udev.USBResource*

path

class labgrid.resource.udev.**IMXUSBLoader**(*target*, *match*, *device=None*)
Bases: *labgrid.resource.udev.USBResource*

filter_match(*device*)

class labgrid.resource.udev.**MXSUSBLoader**(*target*, *match*, *device=None*)
Bases: *labgrid.resource.udev.USBResource*

filter_match(*device*)

class labgrid.resource.udev.**AndroidFastboot**(*target*, *match*, *device=None*)
Bases: *labgrid.resource.udev.USBResource*

filter_match(*device*)

class labgrid.resource.udev.**USBEthernetInterface**(*target*, *match*, *device=None*, *ifname=None*)
Bases: *labgrid.resource.base.EthernetInterface*, *labgrid.resource.udev.USBResource*

update()

if_state

class labgrid.resource.udev.**AlteraUSBBlaster**(*target*, *match*, *device=None*)
Bases: *labgrid.resource.udev.USBResource*

filter_match(*device*)

labgrid.strategy package

Submodules

labgrid.strategy.bareboxstrategy module

exception labgrid.strategy.bareboxstrategy.**StrategyError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

class labgrid.strategy.bareboxstrategy.**Status**
Bases: enum.Enum

An enumeration.

unknown = 0

barebox = 1

shell = 2

class labgrid.strategy.bareboxstrategy.**BareboxStrategy**(*target*, *status=<Status.unknown: 0>*)
Bases: *labgrid.strategy.common.Strategy*

BareboxStrategy - Strategy to switch to barebox or shell

bindings = {'barebox': <class 'labgrid.driver.bareboxdriver.BareboxDriver'>, 'power': <class 'labgrid.protocol.powerp

status = Attribute(name='status', default=<Status.unknown: 0>, validator=None, repr=True, cmp=True, hash=None, in

transition(*status*, *, *step*)

labgrid.strategy.common module

class labgrid.strategy.common.**Strategy**(*target*)
Bases: *labgrid.driver.common.Driver*

Represents a strategy which places a target into a requested state by calling specific drivers. A strategy usually needs to know some details of a given target.

Life cycle: - create - bind (n times) - usage

TODO: This might also be just a driver?

on_client_bound(*client*)

on_activate()

on_deactivate()

resolve_conflicts(*client*)

labgrid.strategy.shellstrategy module

exception labgrid.strategy.shellstrategy.**StrategyError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

class labgrid.strategy.shellstrategy.**Status**
Bases: enum.Enum

An enumeration.

unknown = 0

off = 1

shell = 2

class labgrid.strategy.shellstrategy.**ShellStrategy**(*target*, *status*=<*Status.unknown: 0*>)
Bases: *labgrid.strategy.common.Strategy*

ShellStrategy - Strategy to switch to shell

bindings = {'shell': <class 'labgrid.driver.shelldriver.ShellDriver'>, 'power': <class 'labgrid.protocol.powerprotocol.Po

status = Attribute(name='status', default=<Status.unknown: 0>, validator=None, repr=True, cmp=True, hash=None, i

transition(*status*, *, *step*)

labgrid.strategy.ubootstrategy module

exception labgrid.strategy.ubootstrategy.**StrategyError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, c

class labgrid.strategy.ubootstrategy.**Status**
Bases: enum.Enum

An enumeration.

unknown = 0

uboot = 1

shell = 2

class labgrid.strategy.ubootstrategy.**UBootStrategy**(*target*, *status*=<*Status.unknown: 0*>)
Bases: *labgrid.strategy.common.Strategy*

UbootStrategy - Strategy to switch to uboot or shell

bindings = {'uboot': <class 'labgrid.driver.ubootdriver.UBootDriver'>, 'shell': <class 'labgrid.driver.shelldriver.ShellD

status = Attribute(name='status', default=<Status.unknown: 0>, validator=None, repr=True, cmp=True, hash=None, i

transition(*status*)

labgrid.util package

Submodules

labgrid.util.dict module

labgrid.util.dict.**diff_dict**(*old*, *new*)
Compares old and new dictionaries, yielding for each difference (key, old_value, new_value). None is used for missing values.

labgrid.util.dict.**flat_dict**(*d*)

labgrid.util.exceptions module

exception labgrid.util.exceptions.**NoValidDriverError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

labgrid.util.expect module

class labgrid.util.expect.**PtxExpect**(*driver*, *logfile=None*, *timeout=30*, *cwd=None*)
Bases: pexpect.pty_spawn.spawn

labgrid Wrapper of the pexpect module.

This class provides pexpect functionality for the ConsoleProtocol classes. driver: ConsoleProtocol object to be passed in

send(*s*)
Write to underlying transport, return number of bytes written

read_nonblocking(*size=1*, *timeout=-1*)
Pexpects needs a nonblocking read function, simply use pyserial with a timeout of 0

labgrid.util.marker module

labgrid.util.marker.**gen_marker**()

labgrid.util.qmp module

class labgrid.util.qmp.**QMPMonitor**(*monitor_out*, *monitor_in*)
Bases: object

monitor_out = Attribute(name='monitor_out', default=NOTHING, validator=None, repr=True, cmp=True, hash=Non

monitor_in = Attribute(name='monitor_in', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, i

execute(*command*)

exception labgrid.util.qmp.**QMPError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

labgrid.util.timeout module

class labgrid.util.timeout.**Timeout**(*timeout=120.0*)
Bases: object

Reperents a timeout (as a deadline)

timeout = Attribute(name='timeout', default=120.0, validator=<instance_of validator for type <class 'float'>>, repr=Tru

remaining

expired

labgrid.util.yaml module

labgrid.util.yaml.**load**(*file*)

Submodules

labgrid.binding module

exception labgrid.binding.**StateError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

exception labgrid.binding.**BindingError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

class labgrid.binding.**BindingState**
Bases: enum.Enum

An enumeration.

error = -1

idle = 0

bound = 1

active = 2

class labgrid.binding.**BindingMixin**(*target*)
Bases: object

Handles the binding and activation of drivers and their supplying resources and drivers.

One client can be bound to many suppliers, and one supplier can be bound by many clients.

Conflicting access to one supplier can be avoided by deactivating conflicting clients before activation (using the resolve_conflicts callback).

bindings = {}

target = Attribute(name='target', default=NOTHING, validator=None, repr=True, cmp=True, hash=None, init=True, c

state = Attribute(name='state', default=<BindingState.idle: 0>, validator=None, repr=True, cmp=True, hash=None, ini

on_supplier_bound(*supplier*, *name*)
Called by the Target after a new supplier has been bound

on_client_bound(*client*)
Called by the Target after a new client has been bound

on_activate()
Called by the Target when this object has been activated

on_deactivate()
Called by the Target when this object has been deactivated

resolve_conflicts(*client*)
Called by the Target to allow this object to deactivate conflicting clients.

classmethod check_active(*func*)

labgrid.config module

Config convenience class

This class encapsulates access functions to the environment configuration

class labgrid.config.**Config**(*filename*)
Bases: object

filename = Attribute(name='filename', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, re

resolve_path(*path*)
Resolve an absolute path

Parameters **path** (*str*) – path to resolve

Returns the absolute path

Return type str

get_tool(*tool*)
Retrieve an entry from the tools subkey

Parameters **tool** (*str*) – the tool to retrieve the path for

Returns path to the requested tools

Return type str

get_image_path(*kind*)
Retrieve an entry from the images subkey

Parameters **kind** (*str*) – the kind of the image to retrieve the path for

Returns path to the image

Return type str

Raises KeyError – if the requested image can not be found in the configuration

get_path(*kind*)
Retrieve an entry from the paths subkey

Parameters **kind** (*str*) – the type of path to retrieve the path for

Returns path to the path

Return type str

Raises KeyError – if the requested image can not be found in the configuration

get_option(*name*, *default=None*)
Retrieve an entry from the options subkey

Parameters

- **name** (*str*) – name of the option
- **default** (*str*) – A default parameter in case the option can not be found

Returns value of the option or default parameter

Return type str

Raises KeyError – if the requested image can not be found in the configuration

set_option(*name*, *value*)
Set an entry in the options subkey

Parameters

- **name** (*str*) – name of the option
- **value** (*str*) – the new value

get_targets()

get_imports()
Helper function that returns the list of all imports

Returns List of files which should be imported

Return type *list*

get_paths()
Helper function that returns the subdict of all paths

Returns Dictionary containing all path definitions

Return type Dict

get_images()
Helper function that returns the subdict of all images

Returns Dictionary containing all image definitions

Return type Dict

labgrid.environment module

class labgrid.environment.**Environment**(*config_file='config.yaml'*, *interact=<built-in function input>*)
Bases: object

An environment encapsulates targets.

config_file = Attribute(name='config_file', default='config.yaml', validator=<instance_of validator for type <class 'st

interact = Attribute(name='interact', default=<built-in function input>, validator=None, repr=False, cmp=True, hash=

get_target(*role: str = 'main'*) → labgrid.target.Target
Returns the specified target.

Each target is initialized as needed.

cleanup()

labgrid.exceptions module

exception labgrid.exceptions.**NoConfigFoundError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

exception labgrid.exceptions.**NoSupplierFoundError**(*msg*)
Bases: Exception

msg = Attribute(name='msg', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True, cm

exception labgrid.exceptions.**NoDriverFoundError**(*msg*)
Bases: *labgrid.exceptions.NoSupplierFoundError*

exception labgrid.exceptions.**NoResourceFoundError**(*msg*)
Bases: *labgrid.exceptions.NoSupplierFoundError*

labgrid.factory module

class labgrid.factory.**TargetFactory**
Bases: object

reg_resource(*cls*)
Register a resource with the factory.

Returns the class to allow using it as a decorator.

reg_driver(*cls*)
Register a driver with the factory.

Returns the class to allow using it as a decorator.

make_resource(*target*, *resource*, *args*)

make_target(*name*, *config*, *, *env=None*)

labgrid.factory.**target_factory = <labgrid.factory.TargetFactory object>**
Global TargetFactory instance

This instance is used to register Resource and Driver classes so that Targets can be created automatically from YAML files.

labgrid.step module

class labgrid.step.**Steps**
Bases: object

get_current()

get_new(*title*)

push(*step*)

pop(*step*)

subscribe(*callback*)

notify(*event*)

class labgrid.step.**StepEvent**(*step*, *data*, *, *resource=None*, *stream=False*)
Bases: object

merge(*other*)

age

class labgrid.step.**Step**(*title*, *level*)
Bases: object

duration

status

is_active

is_done

start()

skip(*reason*)

stop()

labgrid.step.**step**(*, *title=None*, *args=[]*, *result=False*)

labgrid.stepreporter module

class labgrid.stepreporter.**StepReporter**
Bases: object

instance = None

classmethod start()

notify(*event*)

labgrid.target module

class labgrid.target.**Target**(*name*, *env=None*)
Bases: object

activate(*client*)
Activate the client by activating all bound suppliers. This may require deactivating other clients.

await_resources(*resources*, *timeout=None*)
Poll the given resources and wait until they are available.

bind(*bindable*)

bind_driver(*client*)
Bind the driver to all suppliers (resources and other drivers).

Currently, we only support binding all suppliers at once.

bind_resource(*resource*)
Bind the resource to this target.

cleanup()
Clean up conntected drivers and resources in reversed order

deactivate(*client*)
Recursively deactivate the client's clients and itself.

This is needed to ensure that no client has an inactive supplier.

env = Attribute(name='env', default=None, validator=None, repr=True, cmp=True, hash=None, init=True, convert=None

get_active_driver(*cls*)
Helper function to get the active driver of the target. Returns the active driver found, otherwise None.

Arguments: cls – driver-class to return as a resource

get_driver(*cls*, *, *activate=True*)
Helper function to get a driver of the target. Returns the first valid driver found, otherwise None.

Arguments: cls – driver-class to return as a resource activate – activate the driver (default True)

get_resource(*cls*, *, *await=True*)
Helper function to get a resource of the target. Returns the first valid resource found, otherwise None.

Arguments: cls – resource-class to return as a resource await – wait for the resource to become available (default True)

interact(*msg*)

name = Attribute(name='name', default=NOTHING, validator=<instance_of validator for type <class 'str'>>, repr=True,

update_resources()

Iterate over all relevant managers and deactivate any active but unavailable resources.

CHAPTER 10

Indices and Tables

- genindex
- modindex
- search

Python Module Index

l

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W