devlib Documentation

Release 1.0.0

ARM Limited

May 24, 2022

CONTENTS

devlib provides an interface for interacting with remote targets, such as development boards, mobile devices, etc. It also provides means of collecting various measurements and traces from such targets.

Contents:

CHAPTER

ONE

OVERVIEW

A *Target* instance serves as the main interface to the target device. There are currently four target interfaces:

- *LinuxTarget* for interacting with Linux devices over SSH.
- *AndroidTarget* for interacting with Android devices over adb.
- *ChromeOsTarget*: for interacting with ChromeOS devices over SSH, and their Android containers over adb.
- *LocalLinuxTarget*: for interacting with the local Linux host.

They all work in more-or-less the same way, with the major difference being in how connection settings are specified; though there may also be a few APIs specific to a particular target type (e.g. *AndroidTarget* exposes methods for working with logcat).

1.1 Acquiring a Target

To create an interface to your device, you just need to instantiate one of the *Target* derivatives listed above, and pass it the right `connection_settings`. Code snippet below gives a typical example of instantiating each of the three target types.

```
from devlib import LocalLinuxTarget, LinuxTarget, AndroidTarget

# Local machine requires no special connection settings.
t1 = LocalLinuxTarget()

# For a Linux device, you will need to provide the normal SSH credentials.
# Both password-based, and key-based authentication is supported (password
# authentication requires sshpass to be installed on your host machine).'
t2 = LinuxTarget(connection_settings={'host': '192.168.0.5',
                                      'username': 'root',
                                      'password': 'sekrit',
                                      # or
                                      'keyfile': '/home/me/.ssh/id_rsa'})
# ChromeOsTarget connection is performed in the same way as LinuxTarget

# For an Android target, you will need to pass the device name as reported
# by "adb devices". If there is only one device visible to adb, you can omit
# this setting and instantiate similar to a local target.
t3 = AndroidTarget(connection_settings={'device': '0123456789abcde'})
```

Instantiating a target may take a second or two as the remote device will be queried to initialize *Target*'s internal state. If you would like to create a *Target* instance but not immediately connect to the remote device, you can pass

`connect=False` parameter. If you do that, you would have to then explicitly call `t.connect()` before you can interact with the device.

There are a few additional parameters you can pass in instantiation besides `connection_settings`, but they are usually unnecessary. Please see *Target* API documentation for more details.

1.2 Target Interface

This is a quick overview of the basic interface to the device. See *Target* API documentation for the full list of supported methods and more detailed documentation.

1.2.1 One-time Setup

```
from devlib import LocalLinuxTarget
t = LocalLinuxTarget()

t.setup()
```

This sets up the target for `devlib` interaction. This includes creating working directories, deploying busybox, etc. It's usually enough to do this once for a new device, as the changes this makes will persist across reboots. However, there is no issue with calling this multiple times, so, to be on the safe side, it's a good idea to call this once at the beginning of your scripts.

1.2.2 Command Execution

There are several ways to execute a command on the target. In each case, an instance of a subclass of `TargetError` will be raised if something goes wrong. When a transient error is encountered such as the loss of the network connectivity, it will raise a `TargetTransientError`. When the command fails, it will raise a `TargetStableError` unless the `will_succeed=True` parameter is specified, in which case a `TargetTransientError` will be raised since it is assumed that the command cannot fail unless there is an environment issue. In each case, it is also possible to specify `as_root=True` if the specified command should be executed as root.

```
from devlib import LocalLinuxTarget
t = LocalLinuxTarget()

# Execute a command
output = t.execute('echo $PWD')

# Execute command via a subprocess and return the corresponding Popen object.
# This will block current connection to the device until the command
# completes.
p = t.background('echo $PWD')
output, error = p.communicate()

# Run the command in the background on the device and return immediately.
# This will not block the connection, allowing to immediately execute another
# command.
t.kick_off('echo $PWD')

# This is used to invoke an executable binary on the device. This allows some
```

(continues on next page)

(continued from previous page)

```
# finer-grained control over the invocation, such as specifying the directory
# in which the executable will run; however you're limited to a single binary
# and cannot construct complex commands (e.g. this does not allow chaining or
# piping several commands together).
output = t.invoke('echo', args=['$PWD'], in_directory='/')
```

1.2.3 File Transfer

```
from devlib import LocalLinuxTarget
t = LocalLinuxTarget()

# "push" a file from the local machine onto the target device.
t.push('/path/to/local/file.txt', '/path/to/target/file.txt')

# "pull" a file from the target device into a location on the local machine
t.pull('/path/to/target/file.txt', '/path/to/local/file.txt')

# Install the specified binary on the target. This will deploy the file and
# ensure it's executable. This will *not* guarantee that the binary will be
# in PATH. Instead the path to the binary will be returned; this should be
# used to call the binary henceforth.
target_bin = t.install('/path/to/local/bin.exe')
# Example invocation:
output = t.execute('{} --some-option'.format(target_bin))
```

The usual access permission constraints on the user account (both on the target and the host) apply.

1.2.4 Process Control

```
import signal
from devlib import LocalLinuxTarget
t = LocalLinuxTarget()

# return PIDs of all running instances of a process
pids = t.get_pids_of('sshd')

# kill a running process. This works the same ways as the kill command, so
# SIGTERM will be used by default.
t.kill(666, signal=signal.SIGKILL)

# kill all running instances of a process.
t.killall('badexe', signal=signal.SIGKILL)

# List processes running on the target. This returns a list of parsed
# PsEntry records.
entries = t.ps()
# e.g.  print virtual memory sizes of all running sshd processes:
print(', '.join(str(e.vsize) for e in entries if e.name == 'sshd'))
```

1.2.5 More...

As mentioned previously, the above is not intended to be exhaustive documentation of the *Target* interface. Please refer to the API documentation for the full list of attributes and methods and their parameters.

1.3 Super User Privileges

It is not necessary for the account logged in on the target to have super user privileges, however the functionality will obviously be diminished, if that is not the case. `devlib` will determine if the logged in user has root privileges and the correct way to invoke it. You should avoid including "sudo" directly in your commands, instead, specify `as_root=True` where needed. This will make your scripts portable across multiple devices and OS's.

1.4 On-Target Locations

File system layouts vary wildly between devices and operating systems. Hard-coding absolute paths in your scripts will mean there is a good chance they will break if run on a different device. To help with this, `devlib` defines a couple of "standard" locations and a means of working with them.

working_directory This is a directory on the target readable and writable by the account used to log in. This should generally be used for all output generated by your script on the device and as the destination for all host-to-target file transfers. It may or may not permit execution so executables should not be run directly from here.

executables_directory This directory allows execution. This will be used by `install()`.

```
from devlib import LocalLinuxTarget
t = LocalLinuxTarget()

# t.path  is equivalent to Python standard library's os.path, and should be
# used in the same way. This insures that your scripts are portable across
# both target and host OS variations. e.g.
on_target_path = t.path.join(t.working_directory, 'assets.tar.gz')
t.push('/local/path/to/assets.tar.gz', on_target_path)

# Since working_directory is a common base path for on-target locations,
# there a short-hand for the above:
t.push('/local/path/to/assets.tar.gz', t.get_workpath('assets.tar.gz'))
```

1.5 Exceptions Handling

Devlib custom exceptions all derive from `DevlibError`. Some exceptions are further categorized into `DevlibTransientError` and `DevlibStableError`. Transient errors are raised when there is an issue in the environment that can happen randomly such as the loss of network connectivity. Even a properly configured environment can be subject to such transient errors. Stable errors are related to either programming errors or configuration issues in the broad sense. This distinction allows quicker analysis of failures, since most transient errors can be ignored unless they happen at an alarming rate. `DevlibTransientError` usually propagates up to the caller of devlib APIs, since it means that an operation could not complete. Retrying it or bailing out is therefore a responsability of the caller.

The hierarchy is as follows:

- `DevlibError`

  - WorkerThreadError
  - HostError
  - TargetError
    * TargetStableError
    * TargetTransientError
    * TargetNotRespondingError
  - DevlibStableError
    * TargetStableError
  - DevlibTransientError
    * TimeoutError
    * TargetTransientError
    * TargetNotRespondingError

1.5.1 Extending devlib

New devlib code is likely to face the decision of raising a transient or stable error. When it is unclear which one should be used, it can generally be assumed that the system is properly configured and therefore, the error is linked to an environment transient failure. If a function is somehow probing a property of a system in the broad meaning, it can use a stable error as a way to signal a non-expected value of that property even if it can also face transient errors. An example are the various `execute()` methods where the command can generally not be assumed to be supposed to succeed by devlib. Their failure does not usually come from an environment random issue, but for example a permission error. The user can use such expected failure to probe the system. Another example is boot completion detection on Android: boot failure cannot be distinguished from a timeout which is too small. A non-transient exception is still raised, since assuming the timeout comes from a network failure would either make the function useless, or force the calling code to handle a transient exception under normal operation. The calling code would potentially wrongly catch transient exceptions raised by other functions as well and attach a wrong meaning to them.

1.6 Modules

Additional functionality is exposed via modules. Modules are initialized as attributes of a target instance. By default, `hotplug`, `cpufreq`, `cpuidle`, `cgroups` and `hwmon` will attempt to load on target; additional modules may be specified when creating a *Target* instance.

A module will probe the target for support before attempting to load. So if the underlying platform does not support particular functionality (e.g. the kernel on target device was built without hotplug support). To check whether a module has been successfully installed on a target, you can use `has()` method, e.g.

```
from devlib import LocalLinuxTarget
t = LocalLinuxTarget()

cpu0_freqs = []
if t.has('cpufreq'):
    cpu0_freqs = t.cpufreq.list_frequencies(0)
```

Please see the modules documentation for more detail.

1.7 Instruments and Collectors

You can retrieve multiple types of data from a target. There are two categories of classes that allow for this:

- An `Instrument` which may be used to collect measurements (such as power) from targets that support it. Please see the *instruments documentation* for more details.
- A `Collector` may be used to collect arbitary data from a `Target` varying from screenshots to trace data. Please see the *collectors documentation* for more details.

An example workflow using `FTraceCollector` is as follows:

```
from devlib import AndroidTarget, FtraceCollector
t = LocalLinuxTarget()

# Initialize a collector specifying the events you want to collect and
# the buffer size to be used.
trace = FtraceCollector(t, events=['power*'], buffer_size=40000)

# As a context manager, clear ftrace buffer using trace.reset(),
# start trace collection using trace.start(), then stop it Using
# trace.stop(). Using a context manager brings the guarantee that
# tracing will stop even if an exception occurs, including
# KeyboardInterrupt (ctr-C) and SystemExit (sys.exit)
with trace:
   # Perform the operations you want to trace here...
   import time; time.sleep(5)

# extract the trace file from the target into a local file
trace.get_data('/tmp/trace.bin')

# View trace file using Kernelshark (must be installed on the host).
trace.view('/tmp/trace.bin')

# Convert binary trace into text format. This would normally be done
# automatically during get_data(), unless autoreport is set to False during
# instantiation of the trace collector.
trace.report('/tmp/trace.bin', '/tmp/trace.txt')
```

CHAPTER

TWO

TARGET

class devlib.target.**Target**(*connection_settings=None*, *platform=None*, *working_directory=None*, *executables_directory=None*, *connect=True*, *modules=None*, *load_default_modules=True*, *shell_prompt=DEFAULT_SHELL_PROMPT*, *conn_cls=None*)

Target is the primary interface to the remote device. All interactions with the device are performed via a *Target* instance, either directly, or via its modules or a wrapper interface (such as an *Instrument*).

Parameters

- **connection_settings** – A `dict` that specifies how to connect to the remote device. Its contents depend on the specific *Target* type (used see *Connection Types*).
- **platform** – A *Target* defines interactions at Operating System level. A *Platform* describes the underlying hardware (such as CPUs available). If a *Platform* instance is not specified on *Target* creation, one will be created automatically and it will dynamically probe the device to discover as much about the underlying hardware as it can. See also *Platform*.
- **working_directory** – This is primary location for on-target file system interactions performed by `devlib`. This location *must* be readable and writable directly (i.e. without sudo) by the connection's user account. It may or may not allow execution. This location will be created, if necessary, during *setup()*.

  If not explicitly specified, this will be set to a default value depending on the type of *Target*
- **executables_directory** – This is the location to which `devlib` will install executable binaries (either during *setup()* or via an explicit *install()* call). This location *must* support execution (obviously). It should also be possible to write to this location, possibly with elevated privileges (i.e. on a rooted Linux target, it should be possible to write here with sudo, but not necessarily directly by the connection's account). This location will be created, if necessary, during *setup()*.

  This location does *not* need to be same as the system's executables location. In fact, to prevent devlib from overwriting system's defaults, it better if this is a separate location, if possible.

  If not explicitly specified, this will be set to a default value depending on the type of *Target*
- **connect** – Specifies whether a connections should be established to the target. If this is set to `False`, then *connect()* must be explicitly called later on before the *Target* instance can be used.
- **modules** – a list of additional modules to be installed. Some modules will try to install by default (if supported by the underlying target). Current default modules are `hotplug`, `cpufreq`, `cpuidle`, `cgroups`, and `hwmon` (See *Modules*).

  See modules documentation for more detail.

- **load_default_modules** – If set to `False`, default modules listed above will *not* attempt to load. This may be used to either speed up target instantiation (probing for initializing modules takes a bit of time) or if there is an issue with one of the modules on a particular device (the rest of the modules will then have to be explicitly specified in the `modules`).
- **shell_prompt** – This is a regular expression that matches the shell prompted on the target. This may be used by some modules that establish auxiliary connections to a target over UART.
- **conn_cls** – This is the type of connection that will be used to communicate with the device.

Target.**core_names**
This is a list containing names of CPU cores on the target, in the order in which they are index by the kernel. This is obtained via the underlying *Platform*.

Target.**core_clusters**
Some devices feature heterogeneous core configurations (such as ARM big.LITTLE). This is a list that maps CPUs onto underlying clusters. (Usually, but not always, clusters correspond to groups of CPUs with the same name). This is obtained via the underlying *Platform*.

Target.**big_core**
This is the name of the cores that are the "big"s in an ARM big.LITTLE configuration. This is obtained via the underlying *Platform*.

Target.**little_core**
This is the name of the cores that are the "little"s in an ARM big.LITTLE configuration. This is obtained via the underlying *Platform*.

Target.**is_connected**
A boolean value that indicates whether an active connection exists to the target device.

Target.**connected_as_root**
A boolean value that indicate whether the account that was used to connect to the target device is "root" (uid=0).

Target.**is_rooted**
A boolean value that indicates whether the connected user has super user privileges on the devices (either is root, or is a sudoer).

Target.**kernel_version**
The version of the kernel on the target device. This returns a `KernelVersion` instance that has separate `version` and `release` fields.

Target.**os_version**
This is a dict that contains a mapping of OS version elements to their values. This mapping is OS-specific.

Target.**hostname**
A string containing the hostname of the target.

Target.**hostid**
A numerical id used to represent the identity of the target.

Note: Currently on 64-bit PowerPC devices this id will always be 0. This is due to the included busybox binary being linked with musl.

Target.**system_id**
A unique identifier for the system running on the target. This identifier is intended to be unique for the combination of hardware, kernel, and file system.

Target.**model**
The model name/number of the target device.

Target.**cpuinfo**
This is a Cpuinfo instance which contains parsed contents of /proc/cpuinfo.

Target.**number_of_cpus**
The total number of CPU cores on the target device.

Target.**config**
A KernelConfig instance that contains parsed kernel config from the target device. This may be None if kernel config could not be extracted.

Target.**user**
The name of the user logged in on the target device.

Target.**conn**
The underlying connection object. This will be None if an active connection does not exist (e.g. if connect=False as passed on initialization and *connect()* has not been called).

Note: a *Target* will automatically create a connection per thread. This will always be set to the connection for the current thread.

Target.**connect**([*timeout*])
Establish a connection to the target. It is usually not necessary to call this explicitly, as a connection gets automatically established on instantiation.

Target.**disconnect**()
Disconnect from target, closing all active connections to it.

Target.**get_connection**([*timeout*])
Get an additional connection to the target. A connection can be used to execute one blocking command at time. This will return a connection that can be used to interact with a target in parallel while a blocking operation is being executed.

This should *not* be used to establish an initial connection; use *connect()* instead.

Note: *Target* will automatically create a connection per thread, so you don't normally need to use this explicitly in threaded code. This is generally useful if you want to perform a blocking operation (e.g. using *background()*) while at the same time doing something else in the same host-side thread.

Target.**setup**([*executables*])
This will perform an initial one-time set up of a device for devlib interaction. This involves deployment of tools relied on the *Target*, creation of working locations on the device, etc.

Usually, it is enough to call this method once per new device, as its effects will persist across reboots. However, it is safe to call this method multiple times. It may therefore be a good practice to always call it once at the beginning of a script to ensure that subsequent interactions will succeed.

Optionally, this may also be used to deploy additional tools to the device by specifying a list of binaries to install in the executables parameter.

Target.**reboot**([*hard*[, *connect*[, *timeout*]]])
Reboot the target device.

Parameters

- **hard** – A boolean value. If True a hard reset will be used instead of the usual soft reset. Hard reset must be supported (usually via a module) for this to work. Defaults to False.

- **connect** – A boolean value. If `True`, a connection will be automatically established to the target after reboot. Defaults to `True`.
- **timeout** – If set, this will be used by various (platform-specific) operations during reboot process to detect if the reboot has failed and the device has hung.

`Target.`**`push`**(*source*, *dest*[, *as_root*, *timeout*, *globbing*])

Transfer a file from the host machine to the target device.

If transfer polling is supported (ADB connections and SSH connections), `poll_transfers` is set in the connection, and a timeout is not specified, the push will be polled for activity. Inactive transfers will be cancelled. (See *Connection Types* for more information on polling).

Parameters

- **source** – path on the host
- **dest** – path on the target
- **as_root** – whether root is required. Defaults to false.
- **timeout** – timeout (in seconds) for the transfer; if the transfer does not complete within this period, an exception will be raised. Leave unset to utilise transfer polling if enabled.
- **globbing** – If `True`, the `source` is interpreted as a globbing pattern instead of being take as-is. If the pattern has multiple matches, `dest` must be a folder (or will be created as such if it does not exists yet).

`Target.`**`pull`**(*source*, *dest*[, *as_root*, *timeout*, *globbing*, *via_temp*])

Transfer a file from the target device to the host machine.

If transfer polling is supported (ADB connections and SSH connections), `poll_transfers` is set in the connection, and a timeout is not specified, the pull will be polled for activity. Inactive transfers will be cancelled. (See *Connection Types* for more information on polling).

Parameters

- **source** – path on the target
- **dest** – path on the host
- **as_root** – whether root is required. Defaults to false.
- **timeout** – timeout (in seconds) for the transfer; if the transfer does not complete within this period, an exception will be raised.
- **globbing** – If `True`, the `source` is interpreted as a globbing pattern instead of being take as-is. If the pattern has multiple matches, `dest` must be a folder (or will be created as such if it does not exists yet).
- **via_temp** – If `True`, copy the file first to a temporary location on the target, and then pull it. This can avoid issues some filesystems, notably paramiko + OpenSSH combination having performance issues when pulling big files from sysfs.

`Target.`**`execute`**(*command*[, *timeout*[, *check_exit_code*[, *as_root*[, *strip_colors*[, *will_succeed*[, *force_locale*]]]]]])

Execute the specified command on the target device and return its output.

Parameters

- **command** – The command to be executed.
- **timeout** – Timeout (in seconds) for the execution of the command. If specified, an exception will be raised if execution does not complete with the specified period.

- **check_exit_code** – If `True` (the default) the exit code (on target) from execution of the command will be checked, and an exception will be raised if it is not `0`.
- **as_root** – The command will be executed as root. This will fail on unrooted targets.
- **strip_colours** – The command output will have colour encodings and most ANSI escape sequences striped out before returning.
- **will_succeed** – The command is assumed to always succeed, unless there is an issue in the environment like the loss of network connectivity. That will make the method always raise an instance of a subclass of `DevlibTransientError` when the command fails, instead of a `DevlibStableError`.
- **force_locale** – Prepend `LC_ALL=<force_locale>` in front of the command to get predictable output that can be more safely parsed. If `None`, no locale is prepended.

`Target.`**background**(*command [, stdout [, stderr [, as_root, [, force_locale [, timeout]]]]*)

Execute the command on the target, invoking it via subprocess on the host. This will return `subprocess.Popen` instance for the command.

Parameters

- **command** – The command to be executed.
- **stdout** – By default, standard output will be piped from the subprocess; this may be used to redirect it to an alternative file handle.
- **stderr** – By default, standard error will be piped from the subprocess; this may be used to redirect it to an alternative file handle.
- **as_root** – The command will be executed as root. This will fail on unrooted targets.
- **force_locale** – Prepend `LC_ALL=<force_locale>` in front of the command to get predictable output that can be more safely parsed. If `None`, no locale is prepended.
- **timeout** – Timeout (in seconds) for the execution of the command. When the timeout expires, `BackgroundCommand.cancel()` is executed to terminate the command.

Note: This **will block the connection** until the command completes.

`Target.`**invoke**(*binary*[, *args*[, *in_directory*[, *on_cpus*[, *as_root*[, *timeout*]]]]])

Execute the specified binary on target (must already be installed) under the specified conditions and return the output.

Parameters

- **binary** – binary to execute. Must be present and executable on the device.
- **args** – arguments to be passed to the binary. The can be either a list or a string.
- **in_directory** – execute the binary in the specified directory. This must be an absolute path.
- **on_cpus** – taskset the binary to these CPUs. This may be a single `int` (in which case, it will be interpreted as the mask), a list of `ints`, in which case this will be interpreted as the list of cpus, or string, which will be interpreted as a comma-separated list of cpu ranges, e.g. `"0,4-7"`.
- **as_root** – Specify whether the command should be run as root
- **timeout** – If this is specified and invocation does not terminate within this number of seconds, an exception will be raised.

Target.**background_invoke**(*binary*[, *args*[, *in_directory*[, *on_cpus*[, *as_root*]]]])
Execute the specified binary on target (must already be installed) as a background task, under the specified conditions and return the `subprocess.Popen` instance for the command.

Parameters

- **binary** – binary to execute. Must be present and executable on the device.
- **args** – arguments to be passed to the binary. The can be either a list or a string.
- **in_directory** – execute the binary in the specified directory. This must be an absolute path.
- **on_cpus** – taskset the binary to these CPUs. This may be a single `int` (in which case, it will be interpreted as the mask), a list of `int`s, in which case this will be interpreted as the list of cpus, or string, which will be interpreted as a comma-separated list of cpu ranges, e.g. `"0,4-7"`.
- **as_root** – Specify whether the command should be run as root

Target.**kick_off**(*command*[, *as_root*])
Kick off the specified command on the target and return immediately. Unlike `background()` this will not block the connection; on the other hand, there is not way to know when the command finishes (apart from calling `ps()`) or to get its output (unless its redirected into a file that can be pulled later as part of the command).

Parameters

- **command** – The command to be executed.
- **as_root** – The command will be executed as root. This will fail on unrooted targets.

Target.**read_value**(*path*[, *kind*])
Read the value from the specified path. This is primarily intended for sysfs/procfs/debugfs etc.

Parameters

- **path** – file to read
- **kind** – Optionally, read value will be converted into the specified kind (which should be a callable that takes exactly one parameter).

Target.**read_int**(*self*, *path*)
Equivalent to `Target.read_value(path, kind=devlib.utils.types.integer)`

Target.**read_bool**(*self*, *path*)
Equivalent to `Target.read_value(path, kind=devlib.utils.types.boolean)`

Target.**write_value**(*path*, *value*[, *verify*])
Write the value to the specified path on the target. This is primarily intended for sysfs/procfs/debugfs etc.

Parameters

- **path** – file to write into
- **value** – value to be written
- **verify** – If `True` (the default) the value will be read back after it is written to make sure it has been written successfully. This due to some sysfs entries silently failing to set the written value without returning an error code.

Target.**revertable_write_value**(*path*, *value*[, *verify*])
Same as *Target.write_value()*, but as a context manager that will write back the previous value on exit.

Target.**batch_revertable_write_value**(*kwargs_list*)
Calls *Target.revertable_write_value()* with all the keyword arguments dictionary given in the list. This is a convenience method to update multiple files at once, leaving them in their original state on exit. If one write fails, all the already-performed writes will be reverted as well.

Target.**read_tree_values**(*path*, *depth=1*, *dictcls=dict*[, *tar*[, *decode_unicode*[, *strip_null_char*]]])
Read values of all sysfs (or similar) file nodes under `path`, traversing up to the maximum depth `depth`.

Returns a nested structure of dict-like objects (`dict`s by default) that follows the structure of the scanned sub-directory tree. The top-level entry has a single item who's key is `path`. If `path` points to a single file, the value of the entry is the value ready from that file node. Otherwise, the value is a dict-line object with a key for every entry under `path` mapping onto its value or further dict-like objects as appropriate.

Although the default behaviour should suit most users, it is possible to encounter issues when reading binary files, or files with colons in their name for example. In such cases, the `tar` parameter can be set to force a full archive of the tree using tar, hence providing a more robust behaviour. This can, however, slow down the read process significantly.

Parameters

- **path** – sysfs path to scan
- **depth** – maximum depth to descend
- **dictcls** – a dict-like type to be used for each level of the hierarchy.
- **tar** – the files will be read using tar rather than grep
- **decode_unicode** – decode the content of tar-ed files as utf-8
- **strip_null_char** – remove null chars from utf-8 decoded files

Target.**read_tree_values_flat**(*path*, *depth=1*)
Read values of all sysfs (or similar) file nodes under `path`, traversing up to the maximum depth `depth`.

Returns a dict mapping paths of file nodes to corresponding values.

Parameters

- **path** – sysfs path to scan
- **depth** – maximum depth to descend

Target.**reset**()
Soft reset the target. Typically, this means executing `reboot` on the target.

Target.**check_responsive**()
Returns `True` if the target appears to be responsive and `False` otherwise.

Target.**kill**(*pid*[, *signal*[, *as_root*]])
Kill a process on the target.

Parameters

- **pid** – PID of the process to be killed.
- **signal** – Signal to be used to kill the process. Defaults to `signal.SIGTERM`.
- **as_root** – If set to `True`, kill will be issued as root. This will fail on unrooted targets.

Target.**killall**(*name*[, *signal*[, *as_root*]])
Kill all processes with the specified name on the target. Other parameters are the same as for `kill()`.

Target.**get_pids_of**(*name*)
Return a list of PIDs of all running instances of the specified process.

Target.**ps**()
Return a list of PsEntry instances for all running processes on the system.

Target.**makedirs**(*self*, *path*)
Create a directory at the given path and all its ancestors if needed.

Target.**file_exists**(*self*, *filepath*)
Returns True if the specified path exists on the target and False otherwise.

Target.**list_file_systems**()
Lists file systems mounted on the target. Returns a list of FstabEntrys.

Target.**list_directory**(*path*[, *as_root*])
List (optionally, as root) the contents of the specified directory. Returns a list of strings.

Target.**get_workpath**(*self*, *path*)
Convert the specified path to an absolute path relative to working_directory on the target. This is a shortcut for t.path.join(t.working_directory, path)

Target.**tempfile**([*prefix*[, *suffix*]])
Get a path to a temporary file (optionally, with the specified prefix and/or suffix) on the target.

Target.**remove**(*path*[, *as_root*])
Delete the specified path on the target. Will work on files and directories.

Target.**core_cpus**(*core*)
Return a list of numeric cpu IDs corresponding to the specified core name.

Target.**list_online_cpus**([*core*])
Return a list of numeric cpu IDs for all online CPUs (optionally, only for CPUs corresponding to the specified core).

Target.**list_offline_cpus**([*core*])
Return a list of numeric cpu IDs for all offline CPUs (optionally, only for CPUs corresponding to the specified core).

Target.**getenv**(*variable*)
Return the value of the specified environment variable on the device

Target.**capture_screen**(*filepath*)
Take a screenshot on the device and save it to the specified file on the host. This may not be supported by the target. You can optionally insert a {ts} tag into the file name, in which case it will be substituted with on-target timestamp of the screen shot in ISO8601 format.

Target.**install**(*filepath*[, *timeout*[, *with_name*]])
Install an executable on the device.

Parameters

- **filepath** – path to the executable on the host
- **timeout** – Optional timeout (in seconds) for the installation
- **with_name** – This may be used to rename the executable on the target

Target.**install_if_needed**(*host_path*, *search_system_binaries=True*)
Check to see if the binary is already installed on the device and if not, install it.

Parameters

- **host_path** – path to the executable on the host
- **search_system_binaries** – Specify whether to search the devices PATH when checking to see if the executable is installed, otherwise only check user installed binaries.

Target.**uninstall**(*name*)
Uninstall the specified executable from the target

Target.**get_installed**(*name*)
Return the full installation path on the target for the specified executable, or None if the executable is not installed.

Target.**which**(*name*)
Alias for get_installed()

Target.**is_installed**(*name*)
Returns True if an executable with the specified name is installed on the target and False other wise.

Target.**extract**(*path*, *dest=None*)
Extracts the specified archive/file and returns the path to the extracted contents. The extraction method is determined based on the file extension. zip, tar, gzip, and bzip2 are supported.

Parameters **dest** –

Specified an on-target destination directory (which must exist) for the extracted contents.

Returns the path to the extracted contents. In case of files (gzip and bzip2), the path to the decompressed file is returned; for archives, the path to the directory with the archive's contents is returned.

Target.**is_network_connected**()
Checks for internet connectivity on the device. This doesn't actually guarantee that the internet connection is "working" (which is rather nebulous), it's intended just for failing early when definitively _not_ connected to the internet.

Returns True if internet seems available, False otherwise.

Target.**install_module**(*mod*, ***params*)

Parameters

- **mod** – The module name or object to be installed to the target.
- **params** – Keyword arguments used to instantiate the module.

Installs an additional module to the target after the initial setup has been performed.

2.1 Linux Target

class devlib.target.**LinuxTarget**(*connection_settings=None*, *platform=None*, *working_directory=None*, *executables_directory=None*, *connect=True*, *modules=None*, *load_default_modules=True*, *shell_prompt=DEFAULT_SHELL_PROMPT*, *conn_cls=SshConnection*, *is_container=False*)
LinuxTarget is a subclass of *Target* with customisations specific to a device running linux.

2.2 Local Linux Target

class devlib.target.**LocalLinuxTarget**(*connection_settings=None*, *platform=None*, *working_directory=None*, *executables_directory=None*, *connect=True*, *modules=None*, *load_default_modules=True*, *shell_prompt=DEFAULT_SHELL_PROMPT*, *conn_cls=SshConnection*, *is_container=False*)

`LocalLinuxTarget` is a subclass of `LinuxTarget` with customisations specific to using the host machine running linux as the target.

2.3 Android Target

class devlib.target.**AndroidTarget**(*connection_settings=None*, *platform=None*, *working_directory=None*, *executables_directory=None*, *connect=True*, *modules=None*, *load_default_modules=True*, *shell_prompt=DEFAULT_SHELL_PROMPT*, *conn_cls=AdbConnection*, *package_data_directory='/data/data'*)

`AndroidTarget` is a subclass of `Target` with additional features specific to a device running Android.

Parameters **package_data_directory** – This is the location of the data stored for installed Android packages on the device.

AndroidTarget.**set_rotation**(*rotation*)
Specify an integer representing the desired screen rotation with the following mappings: Natural: `0`, Rotated Left: `1`, Inverted : `2` and Rotated Right : `3`.

AndroidTarget.**get_rotation**(*rotation*)
Returns an integer value representing the orientation of the devices screen. `0` : Natural, `1` : Rotated Left, `2` : Inverted and `3` : Rotated Right.

AndroidTarget.**set_natural_rotation**()
Sets the screen orientation of the device to its natural (0 degrees) orientation.

AndroidTarget.**set_left_rotation**()
Sets the screen orientation of the device to 90 degrees.

AndroidTarget.**set_inverted_rotation**()
Sets the screen orientation of the device to its inverted (180 degrees) orientation.

AndroidTarget.**set_right_rotation**()
Sets the screen orientation of the device to 270 degrees.

AndroidTarget.**set_auto_rotation**(*autorotate*)
Specify a boolean value for whether the devices auto-rotation should be enabled.

AndroidTarget.**get_auto_rotation**()
Returns `True` if the targets auto rotation is currently enabled and `False` otherwise.

AndroidTarget.**set_airplane_mode**(*mode*)
Specify a boolean value for whether the device should be in airplane mode.

Note: Requires the device to be rooted if the device is running Android 7+.

AndroidTarget.**get_airplane_mode**()
Returns `True` if the target is currently in airplane mode and `False` otherwise.

AndroidTarget.**set_brightness**(*value*)
Sets the devices screen brightness to a specified integer between 0 and 255.

AndroidTarget.**get_brightness**()
Returns an integer between 0 and 255 representing the devices current screen brightness.

AndroidTarget.**set_auto_brightness**(*auto_brightness*)
Specify a boolean value for whether the devices auto brightness should be enabled.

AndroidTarget.**get_auto_brightness**()
Returns True if the targets auto brightness is currently enabled and False otherwise.

AndroidTarget.**set_stay_on_never**()
Sets the stay-on mode to 0, where the screen will turn off as standard after the timeout.

AndroidTarget.**set_stay_on_while_powered**()
Sets the stay-on mode to 7, where the screen will stay on while the device is charging

AndroidTarget.**set_stay_on_mode**(*mode*)
Sets the stay-on mode to the specified number between 0 and 7 (inclusive).

AndroidTarget.**get_stay_on_mode**()
Returns an integer between 0 and 7 representing the current stay-on mode of the device.

AndroidTarget.**ensure_screen_is_off**(*verify=True*)
Checks if the devices screen is on and if so turns it off. If verify is set to True then a TargetStableError will be raise if the display cannot be turned off. E.g. if always on mode is enabled.

AndroidTarget.**ensure_screen_is_on**(*verify=True*)
Checks if the devices screen is off and if so turns it on. If verify is set to True then a TargetStableError will be raise if the display cannot be turned on.

AndroidTarget.**ensure_screen_is_on_and_stays**(*verify=True*, *mode=7*)
Calls AndroidTarget.ensure_screen_is_on(verify) then additionally sets the screen stay on mode to mode.

AndroidTarget.**is_screen_on**()
Returns True if the targets screen is currently on and False otherwise. If the display is in a "Doze" mode or similar always on state, this will return True.

AndroidTarget.**wait_for_device**(*timeout=30*)
Returns when the devices becomes available withing the given timeout otherwise returns a TimeoutError.

AndroidTarget.**reboot_bootloader**(*timeout=30*)
Attempts to reboot the target into it's bootloader.

AndroidTarget.**homescreen**()
Returns the device to its home screen.

AndroidTarget.**swipe_to_unlock**(*direction='diagonal'*)
Performs a swipe input on the device to try and unlock the device. A direction of "horizontal", "vertical" or "diagonal" can be supplied to specify in which direction the swipe should be performed. By default "diagonal" will be used to try and support the majority of newer devices.

2.4 ChromeOS Target

class devlib.target.**ChromeOsTarget**(*connection_settings=None*, *platform=None*, *working_directory=None*, *executables_directory=None*, *android_working_directory=None*, *android_executables_directory=None*, *connect=True*, *modules=None*, *load_default_modules=True*, *shell_prompt=DEFAULT_SHELL_PROMPT*, *package_data_directory='/data/data'*)

ChromeOsTarget is a subclass of *LinuxTarget* with additional features specific to a device running ChromeOS for example, if supported, its own android container which can be accessed via the `android_container` attribute. When making calls to or accessing properties and attributes of the ChromeOS target, by default they will be applied to Linux target as this is where the majority of device configuration will be performed and if not available, will fall back to using the android container if available. This means that all the available methods from *LinuxTarget* and *AndroidTarget* are available for *ChromeOsTarget* if the device supports android otherwise only the *LinuxTarget* methods will be available.

Parameters

- **working_directory** – This is the location of the working directory to be used for the Linux target container. If not specified will default to `"/mnt/stateful_partition/devlib-target"`.
- **android_working_directory** – This is the location of the working directory to be used for the android container. If not specified it will use the working directory default for `AndroidTarget.`.
- **android_executables_directory** – This is the location of the executables directory to be used for the android container. If not specified will default to a `bin` subdirectory in the `android_working_directory.`
- **package_data_directory** – This is the location of the data stored for installed Android packages on the device.

CHAPTER

THREE

MODULES

Modules add additional functionality to the core *Target* interface. Usually, it is support for specific subsystems on the target. Modules are instantiated as attributes of the *Target* instance.

3.1 hotplug

Kernel `hotplug` subsystem allows offlining ("removing") cores from the system, and onlining them back in. The `devlib` module exposes a simple interface to this subsystem

```
from devlib import LocalLinuxTarget
target = LocalLinuxTarget()

# offline cpus 2 and 3, "removing" them from the system
target.hotplug.offline(2, 3)

# bring CPU 2 back in
target.hotplug.online(2)

# Make sure all cpus are online
target.hotplug.online_all()
```

3.2 cpufreq

`cpufreq` is the kernel subsystem for managing DVFS (Dynamic Voltage and Frequency Scaling). It allows controlling frequency ranges and switching policies (governors). The `devlib` module exposes the following interface

---

Note: On ARM big.LITTLE systems, all cores on a cluster (usually all cores of the same type) are in the same frequency domain, so setting `cpufreq` state on one core on a cluster will affect all cores on that cluster. Because of this, some devices only expose cpufreq sysfs interface (which is what is used by the `devlib` module) on the first cpu in a cluster. So to keep your scripts portable, always use the fist (online) CPU in a cluster to set `cpufreq` state.

`target.cpufreq.list_governors(cpu)`
: List cpufreq governors available for the specified cpu. Returns a list of strings.

  Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or `"cpu1"`).

`target.cpufreq.list_governor_tunables(cpu)`
: List the tunables for the specified cpu's current governor.

Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").

target.cpufreq.**get_governor**(*cpu*)
Returns the name of the currently set governor for the specified cpu.

Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").

target.cpufreq.**set_governor**(*cpu*, *governor*, ***kwargs*)
Sets the governor for the specified cpu.

Parameters

- **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").
- **governor** – The name of the governor. This must be one of the governors supported by the CPU (as returned by `list_governors()`.

Keyword arguments may be used to specify governor tunable values.

target.cpufreq.**get_governor_tunables**(*cpu*)
Return a dict with the values of the specified CPU's current governor.

Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").

target.cpufreq.**set_governor_tunables**(*cpu*, ***kwargs*)
Set the tunables for the current governor on the specified CPU.

Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").

Keyword arguments should be used to specify tunable values.

target.cpufreq.**list_frequencies**(*cpu*)
List DVFS frequencies supported by the specified CPU. Returns a list of ints.

Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").

target.cpufreq.**get_min_frequency**(*cpu*)
target.cpufreq.**get_max_frequency**(*cpu*)
target.cpufreq.**set_min_frequency**(*cpu*, *frequency*[, *exact=True*])
target.cpufreq.**set_max_frequency**(*cpu*, *frequency*[, *exact=True*])
Get the currently set, or set new min and max frequencies for the specified CPU. "set" functions are available with all governors other than `userspace`.

Parameters **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").

target.cpufreq.**get_min_available_frequency**(*cpu*)
target.cpufreq.**get_max_available_frequency**(*cpu*)

Retrieve the min or max DVFS frequency that is supported (as opposed to currently enforced) for a given CPU. Returns an int or None if could not be determined.

Parameters **frequency** – Frequency to set.

target.cpufreq.**get_frequency**(*cpu*)
target.cpufreq.**set_frequency**(*cpu*, *frequency*[, *exact=True*])
Get and set current frequency on the specified CPU. `set_frequency` is only available if the current governor is `userspace`.

Parameters

- **cpu** – The cpu; could be a numeric or the corresponding string (e.g. 1 or "cpu1").
- **frequency** – Frequency to set.

3.3 cpuidle

`cpuidle` is the kernel subsystem for managing CPU low power (idle) states.

`target.cpuidle.`**`get_driver`**`()`
Return the name current cpuidle driver.

`target.cpuidle.`**`get_governor`**`()`
Return the name current cpuidle governor (policy).

`target.cpuidle.`**`get_states`**`([`*cpu=0*`])`
Return idle states (optionally, for the specified CPU). Returns a list of `CpuidleState` instances.

`target.cpuidle.`**`get_state`**`(`*state*`[`*, cpu=0*`])`
Return `CpuidleState` instance (optionally, for the specified CPU) representing the specified idle state. `state` can be either an integer index of the state or a string with the states `name` or `desc`.

`target.cpuidle.`**`enable`**`(`*state*`[`*, cpu=0*`])`
`target.cpuidle.`**`disable`**`(`*state*`[`*, cpu=0*`])`
`target.cpuidle.`**`enable_all`**`([`*cpu=0*`])`
`target.cpuidle.`**`disable_all`**`([`*cpu=0*`])`
Enable or disable the specified or all states (optionally on the specified CPU.

You can also call `enable()` or `disable()` on `CpuidleState` objects returned by get_state(s).

3.4 cgroups

TODO

3.5 hwmon

TODO

3.6 API

3.6.1 Generic Module API Description

Modules implement discrete, optional pieces of functionality ("optional" in the sense that the functionality may or may not be present on the target device, or that it may or may not be necessary for a particular application).

Every module (ultimately) derives from `devlib.module.Module` class. A module must define the following class attributes:

name A unique name for the module. This cannot clash with any of the existing names and must be a valid Python identifier, but is otherwise free-form.

kind This identifies the type of functionality a module implements, which in turn determines the interface implemented by the module (all modules of the same kind must expose a consistent interface). This must be a valid Python identifier, but is otherwise free-form, though, where possible, one should try to stick to an already-defined kind/interface, lest we end up with a bunch of modules implementing similar functionality but exposing slightly different interfaces.

> **Note:** It is possible to omit `kind` when defining a module, in which case the module's `name` will be treated as its `kind` as well.

stage This defines when the module will be installed into a *`Target`*. Currently, the following values are allowed:

connected The module is installed after a connection to the target has been established. This is the default.

early The module will be installed when a *`Target`* is first created. This should be used for modules that do not rely on a live connection to the target.

setup The module will be installed after initial setup of the device has been performed. This allows the module to utilize assets deployed during the setup stage for example 'Busybox'.

Additionally, a module must implement a static (or class) method `probe()`:

`Module.`**`probe`**`(`*target*`)`
This method takes a *`Target`* instance and returns `True` if this module is supported by that target, or `False` otherwise.

> **Note:** If the module `stage` is `"early"`, this method cannot assume that a connection has been established (i.e. it can only access attributes of the Target that do not rely on a connection).

Installation and invocation

The default installation method will create an instance of a module (the *`Target`* instance being the sole argument) and assign it to the target instance attribute named after the module's `kind` (or `name` if `kind` is `None`).

It is possible to change the installation procedure for a module by overriding the default `install()` method. The method must have the following signature:

`Module.`**`install`**`(`*cls*, *target*, ***kwargs*`)`
Install the module into the target instance.

Implementation and Usage Patterns

There are two common ways to implement the above API, corresponding to the two common uses for modules:

- If a module provides an interface to a particular set of functionality (e.g. an OS subsystem), that module would typically derive directly form `Module` and would leave `kind` unassigned, so that it is accessed by it name. Its instance's methods and attributes provide the interface for interacting with its functionality. For examples of this type of module, see the subsystem modules listed above (e.g. `cpufreq`).
- If a module provides a platform- or infrastructure-specific implementation of a common function, the module would derive from one of `Module` subclasses that define the interface for that function. In that case the module would be accessible via the common `kind` defined its super. The module would typically implement *`__call__()`* and be invoked directly. For examples of this type of module, see common function interface definitions below.

3.6.2 Common Function Interfaces

This section documents `Module` classes defining interface for common functions. Classes derived from them provide concrete implementations for specific platforms.

HardResetModule

`HardResetModule.`**`kind`**
: "hard_reset"

`HardResetModule.`**`__call__`**`()`
: Must be implemented by derived classes.

  Implements hard reset for a target devices. The equivalent of physically power cycling the device. This may be used by client code in situations where the target becomes unresponsive and/or a regular reboot is not possible.

BootModule

`BootModule.`**`kind`**
: "hard_reset"

`BootModule.`**`__call__`**`()`
: Must be implemented by derived classes.

  Implements a boot procedure. This takes the device from (hard or soft) reset to a booted state where the device is ready to accept connections. For a lot of commercial devices the process is entirely automatic, however some devices (e.g. development boards), my require additional steps, such as interactions with the bootloader, in order to boot into the OS.

`Bootmodule.`**`update`**`(\*\*`*`kwargs`*`)`
: Update the boot settings. Some boot sequences allow specifying settings that will be utilized during boot (e.g. linux kernel boot command line). The default implementation will set each setting in `kwargs` as an attribute of the boot module (or update the existing attribute).

FlashModule

`FlashModule.`**`kind`**
: "flash"

`devlib.module.hwmon.`**`__call__`**`(`*`image_bundle=None`*`, `*`images=None`*`, `*`boot_config=None`*`, `*`connect=True`*`)`
: Must be implemented by derived classes.

  Flash the target platform with the specified images.

  Parameters

  - **`image_bundle`** – A compressed bundle of image files with any associated metadata. The format of the bundle is specific to a particular implementation.
  - **`images`** – A dict mapping image names/identifiers to the path on the host file system of the corresponding image file. If both this and `image_bundle` are specified, individual images will override those in the bundle.
  - **`boot_config`** – Some platforms require specifying boot arguments at the time of flashing the images, rather than during each reboot. For other platforms, this will be ignored.

  Connect Specifiy whether to try and connect to the target after flashing.

3.6.3 Module Registration

Modules are specified on *Target* or *Platform* creation by name. In order to find the class associated with the name, the module needs to be registered with `devlib`. This is accomplished by passing the module class into `register_module()` method once it is defined.

Note: If you're wiring a module to be included as part of `devlib` code base, you can place the file with the module class under `devlib/modules/` in the source and it will be automatically enumerated. There is no need to explicitly register it in that case.

The code snippet below illustrates an implementation of a hard reset function for an "Acme" device.

```
import os
from devlib import HardResetModule, register_module


class AcmeHardReset(HardResetModule):

    name = 'acme_hard_reset'

    def __call__(self):
        # Assuming Acme board comes with a "reset-acme-board" utility
        os.system('reset-acme-board {}'.format(self.target.name))

register_module(AcmeHardReset)
```

CHAPTER

FOUR

INSTRUMENTATION

The `Instrument` API provide a consistent way of collecting measurements from a target. Measurements are collected via an instance of a class derived from *Instrument*. An `Instrument` allows collection of measurement from one or more channels. An `Instrument` may support `INSTANTANEOUS` or `CONTINUOUS` collection, or both.

4.1 Example

The following example shows how to use an instrument to read temperature from an Android target.

```
# import and instantiate the Target and the instrument
# (note: this assumes exactly one android target connected
#  to the host machine).
In [1]: from devlib import AndroidTarget, HwmonInstrument

In [2]: t = AndroidTarget()

In [3]: i = HwmonInstrument(t)

# Set up the instrument on the Target. In case of HWMON, this is
# a no-op, but is included here for completeness.
In [4]: i.setup()

# Find out what the instrument is capable collecting from the
# target.
In [5]: i.list_channels()
Out[5]:
[CHAN(battery/temp1, battery_temperature),
 CHAN(exynos-therm/temp1, exynos-therm_temperature)]

# Set up a new measurement session, and specify what is to be
# collected.
In [6]: i.reset(sites=['exynos-therm'])

# HWMON instrument supports INSTANTANEOUS collection, so invoking
# take_measurement() will return a list of measurements take from
# each of the channels configured during reset()
In [7]: i.take_measurement()
Out[7]: [exynos-therm_temperature: 36.0 degrees]
```

4.2 API

4.2.1 Instrument

class devlib.instrument.**Instrument**(*target*, ***kwargs*)

An `Instrument` allows collection of measurement from one or more channels. An `Instrument` may support `INSTANTANEOUS` or `CONTINUOUS` collection, or both.

Instrument.**mode**

A bit mask that indicates collection modes that are supported by this instrument. Possible values are:

INSTANTANEOUS The instrument supports taking a single sample via `take_measurement()`.

CONTINUOUS The instrument supports collecting measurements over a period of time via `start()`, `stop()`, `get_data()`, and (optionally) `get_raw` methods.

Note: It's possible for one instrument to support more than a single mode.

Instrument.**active_channels**

Channels that have been activated via `reset()`. Measurements will only be collected for these channels.

Instrument.**list_channels**()

Returns a list of *InstrumentChannel* instances that describe what this instrument can measure on the current target. A channel is a combination of a `kind` of measurement (power, temperature, etc) and a `site` that indicates where on the target the measurement will be collected from.

Instrument.**get_channels**(*measure*)

Returns channels for a particular `measure` type. A `measure` can be either a string (e.g. `"power"`) or a `MeasurmentType` instance.

Instrument.**setup**(**args*, ***kwargs*)

This will set up the instrument on the target. Parameters this method takes are particular to subclasses (see documentation for specific instruments below). What actions are performed by this method are also instrument-specific. Usually these will be things like installing executables, starting services, deploying assets, etc. Typically, this method needs to be invoked at most once per reboot of the target (unless `teardown()` has been called), but see documentation for the instrument you're interested in.

Instrument.**reset**(*sites=None*, *kinds=None*, *channels=None*)

This is used to configure an instrument for collection. This must be invoked before `start()` is called to begin collection. This methods sets the `active_channels` attribute of the `Instrument`.

If `channels` is provided, it is a list of names of channels to enable and `sites` and `kinds` must both be `None`.

Otherwise, if one of `sites` or `kinds` is provided, all channels matching the given sites or kinds are enabled. If both are provided then all channels of the given kinds at the given sites are enabled.

If none of `sites`, `kinds` or `channels` are provided then all available channels are enabled.

Instrument.**take_measurement**()

Take a single measurement from `active_channels`. Returns a list of `Measurement` objects (one for each active channel).

Note: This method is only implemented by *Instrument*s that support `INSTANTANEOUS` measurement.

Instrument.**start**()

Starts collecting measurements from `active_channels`.

Note: This method is only implemented by *Instrument*s that support CONTINUOUS measurement.

Instrument.**stop**()
Stops collecting measurements from active_channels. Must be called after *start()*.

Note: This method is only implemented by *Instrument*s that support CONTINUOUS measurement.

Instrument.**get_data**(*outfile*)
Write collected data into outfile. Must be called after *stop()*. Data will be written in CSV format with a column for each channel and a row for each sample. Column heading will be channel, labels in the form <site>_<kind> (see *InstrumentChannel*). The order of the columns will be the same as the order of channels in Instrument.active_channels.

If reporting timestamps, one channel must have a site named "timestamp" and a kind of a MeasurmentType of an appropriate time unit which will be used, if appropriate, during any post processing.

Note: Currently supported time units are seconds, milliseconds and microseconds, other units can also be used if an appropriate conversion is provided.

This returns a MeasurementCsv instance associated with the outfile that can be used to stream Measurements lists (similar to what is returned by take_measurement().

Note: This method is only implemented by *Instrument*s that support CONTINUOUS measurement.

Instrument.**get_raw**()

Returns a list of paths to files containing raw output from the underlying source(s) that is used to produce the data CSV. If no raw output is generated or saved, an empty list will be returned. The format of the contents of the raw files is entirely source-dependent.

Note: This method is not guaranteed to return valid filepaths after the *teardown()* method has been invoked as the raw files may have been deleted. Please ensure that copies are created manually prior to calling *teardown()* if the files are to be retained.

Instrument.**teardown**()
Performs any required clean up of the instrument. This usually includes removing temporary and raw files (if keep_raw is set to False on relevant instruments), stopping services etc.

Instrument.**sample_rate_hz**
Sample rate of the instrument in Hz. Assumed to be the same for all channels.

Note: This attribute is only provided by *Instrument*s that support CONTINUOUS measurement.

4.2.2 Instrument Channel

class devlib.instrument.**InstrumentChannel**(*name*, *site*, *measurement_type*, ***attrs*)

An *InstrumentChannel* describes a single type of measurement that may be collected by an *Instrument*. A channel is primarily defined by a `site` and a `measurement_type`.

A `site` indicates where on the target a measurement is collected from (e.g. a voltage rail or location of a sensor).

A `measurement_type` is an instance of `MeasurmentType` that describes what sort of measurement this is (power, temperature, etc). Each measurement type has a standard unit it is reported in, regardless of an instrument used to collect it.

A channel (i.e. site/measurement_type combination) is unique per instrument, however there may be more than one channel associated with one site (e.g. for both voltage and power).

It should not be assumed that any site/measurement_type combination is valid. The list of available channels can queried with *Instrument.list_channels()*.

InstrumentChannel.**site**

The name of the "site" from which the measurements are collected (e.g. voltage rail, sensor, etc).

InstrumentChannel.**kind**

A string indicating the type of measurement that will be collected. This is the `name` of the `MeasurmentType` associated with this channel.

InstrumentChannel.**units**

Units in which measurement will be reported. this is determined by the underlying `MeasurmentType`.

InstrumentChannel.**label**

A label that can be attached to measurements associated with with channel. This is constructed with

```
'{}_{}'.format(self.site, self.kind)
```

4.2.3 Measurement Types

In order to make instruments easer to use, and to make it easier to swap them out when necessary (e.g. change method of collecting power), a number of standard measurement types are defined. This way, for example, power will always be reported as "power" in Watts, and never as "pwr" in milliWatts. Currently defined measurement types are

name	units	category
count	count	
percent	percent	
time_us	microseconds	time
time_ms	milliseconds	time
temperature	degrees	thermal
power	watts	power/energy
voltage	volts	power/energy
current	amps	power/energy
energy	joules	power/energy
tx	bytes	data transfer
rx	bytes	data transfer
tx/rx	bytes	data transfer

4.3 Available Instruments

This section lists instruments that are currently part of devlib.

Todo: Add other instruments

4.3.1 Baylibre ACME BeagleBone Black Cape

From the official project page:

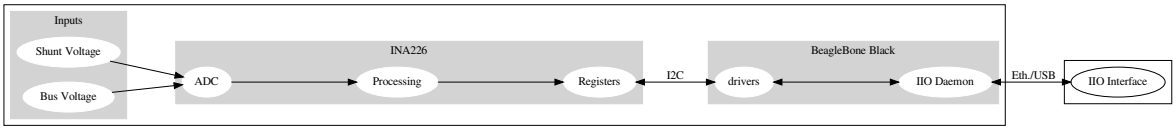
> [The Baylibre Another Cute Measurement Equipment (ACME)] is an extension for the BeagleBone Black (the ACME Cape), designed to provide multi-channel power and temperature measurements capabilities to the BeagleBone Black (BBB). It comes with power and temperature probes integrating a power switch (the ACME Probes), turning it into an advanced all-in-one power/temperature measurement solution.
>
> The ACME initiative is completely open source, from HW to SW drivers and applications.

The Infrastructure

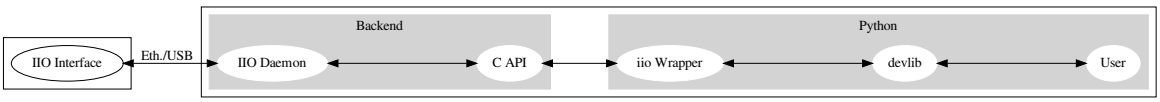
Retrieving measurement from the ACME through devlib requires:

- a BBB running the image built for using the ACME (micro SD card required);
- an ACME cape on top of the BBB;
- at least one ACME probe[1] connected to the ACME cape;
- a BBB-host interface (typically USB or Ethernet)[2];
- a host (the one running devlib) with libiio (the Linux IIO interface) installed, and a Python environment able to find the libiio Python wrapper *i.e.* able to `import iio` as communications between the BBB and the host rely on the Linux Industrial I/O Subsystem (IIO).

The ACME probes are built on top of the Texas Instruments INA226 and the data acquisition chain is as follows:



For reference, the software stack on the host is roughly given by:



Ethernet was the only IIO Interface used and tested during the development of this instrument. However, USB seems to be supported. The IIO library also provides "Local" and "XML" connections but these are to be used when the IIO

[1] There exist different variants of the ACME probe (USB, Jack, shunt resistor) but they all use the same probing hardware (the TI INA226) and don't differ from the point of view of the software stack (at any level, including devlib, the highest one)

[2] Be careful that in cases where multiple ACME boards are being used, it may be required to manually handle name conflicts

devices are directly connected to the host *i.e.* in our case, if we were to run Python and devlib on the BBB. These are also untested.

Measuring Power

In IIO terminology, the ACME cape is an *IIO context* and ACME probes are *IIO devices* with *IIO channels*. An input *IIO channel* (the ACME has no *output IIO channel*) is a stream of samples and an ACME cape can be connected to up to 8 probes *i.e.* have 8 *IIO devices*. The probes are discovered at startup by the IIO drivers on the BBB and are indexed according to the order in which they are connected to the ACME cape (with respect to the "Probe *X*" connectors on the cape).

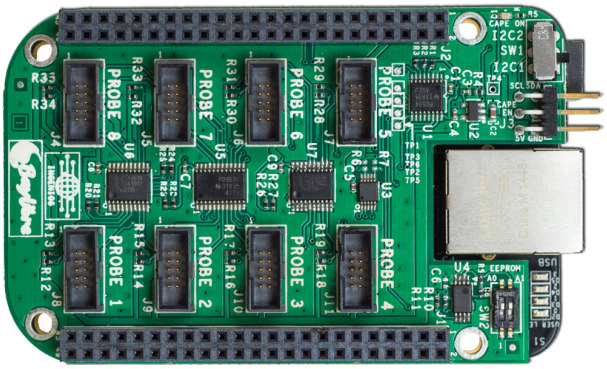


Fig. 1: ACME Cape on top of a BBB: Notice the numbered probe connectors (source)

Please note that the numbers on the PCB do not represent the index of a probe in IIO; on top of being 1-based (as opposed to IIO device indexing being 0-based), skipped connectors do not result in skipped indices *e.g.* if three probes are connected to the cape at `Probe 1`, `Probe 3` and `Probe 7`, IIO (and therefore the entire software stack, including devlib) will still refer to them as devices `0`, `1` and `2`, respectively. Furthermore, probe "hot swapping" does not seem to be supported.

INA226: The probing spearhead

An ACME probe has 5 *IIO channels*, 4 of which being "IIO wrappers" around what the INA226 outputs (through its I2C registers): the bus voltage, the shunt voltage, the shunt current and the load power. The last channel gives the timestamps and is probably added further down the pipeline. A typical circuit configuration for the INA226 (useful when shunt-based ACME probes are used as their PCB does not contain the full circuit unlike the USB and jack variants) is given by its datasheet:

The analog-to-digital converter (ADC)

The digital time-discrete sampled signal of the analog time-continuous input voltage signal is obtained through an analog-to-digital converter (ADC). To measure the "instantaneous input voltage", the ADC "charges up or down" a capacitor before measuring its charge.

The *integration time* is the time spend by the ADC acquiring the input signal in its capacitor. The longer this time is, the more resilient the sampling process is to unwanted noise. The drawback is that, if the integration time is increased then the sampling rate decreases. This effect can be somewhat compared to a *low-pass filter*.

As the INA226 alternatively connects its ADC to the bus voltage and shunt voltage (see previous figure), samples are

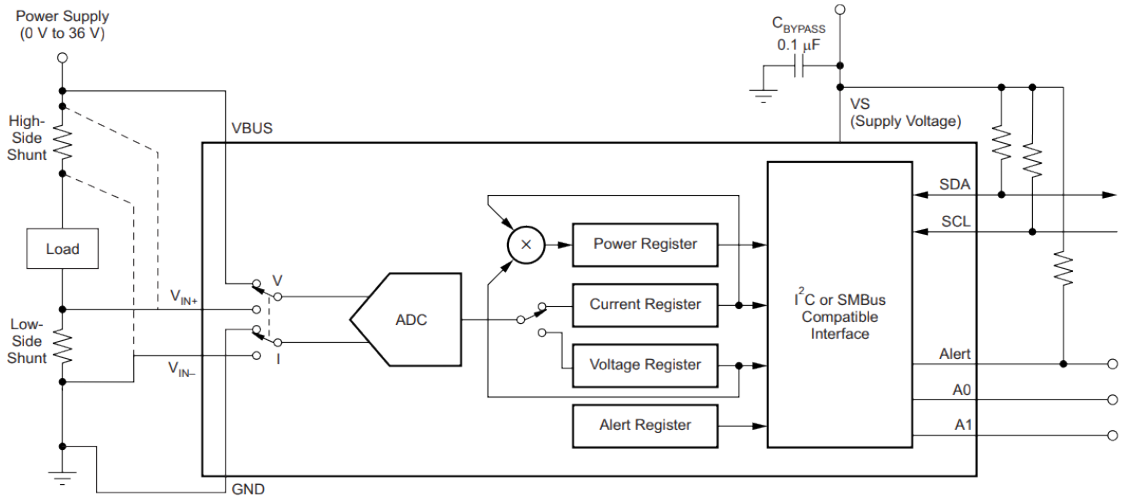


Fig. 2: Typical Circuit Configuration (source: Texas Instruments INA226)

retrieved at a frequency of

$$\frac{1}{T_{bus} + T_{shunt}}$$

where T_X is the integration time for the X voltage.

As described below (`BaylibreAcmeInstrument.reset`), the integration times for the bus and shunt voltage can be set separately which allows a tradeoff of accuracy between signals. This is particularly useful as the shunt voltage returned by the INA226 has a higher resolution than the bus voltage (2.5 V and 1.25 mV LSB, respectively) and therefore would benefit more from a longer integration time.

As an illustration, consider the following sampled sine wave and notice how increasing the integration time (of the bus voltage in this case) "smoothes" out the signal:

Internal signal processing

The INA226 is able to accumulate samples acquired by its ADC and output to the ACME board (technically, to its I2C registers) the average value of N samples. This is called *oversampling*. While the integration time somewhat behaves as an analog low-pass filter, the oversampling feature is a digital low-pass filter by definition. The former should be set to reduce sampling noise (*i.e.* noise on a single sample coming from the sampling process) while the latter should be used to filter out high-frequency noise present in the input signal and control the sampling frequency.

Therefore, samples are available at the output of the INA226 at a frequency

$$\frac{1}{N(T_{bus} + T_{shunt})}$$

and oversampling ratio provides a way to control the output sampling frequency (*i.e.* to limit the required output bandwidth) while making sure the signal fidelity is as desired.

The 4 IIO channels coming from the INA226 can be grouped according to their respective origins: the bus and shunt voltages are measured (and, potentially filtered) while the shunt current and load power are computed. Indeed, the INA226 contains on-board fixed-point arithmetic units to compute the trivial expressions:

$$I_{shunt} = \frac{V_{shunt}}{R_{shunt}}, \quad P_{load} = V_{load}\, I_{load} \approx V_{bus}\, I_{shunt}$$

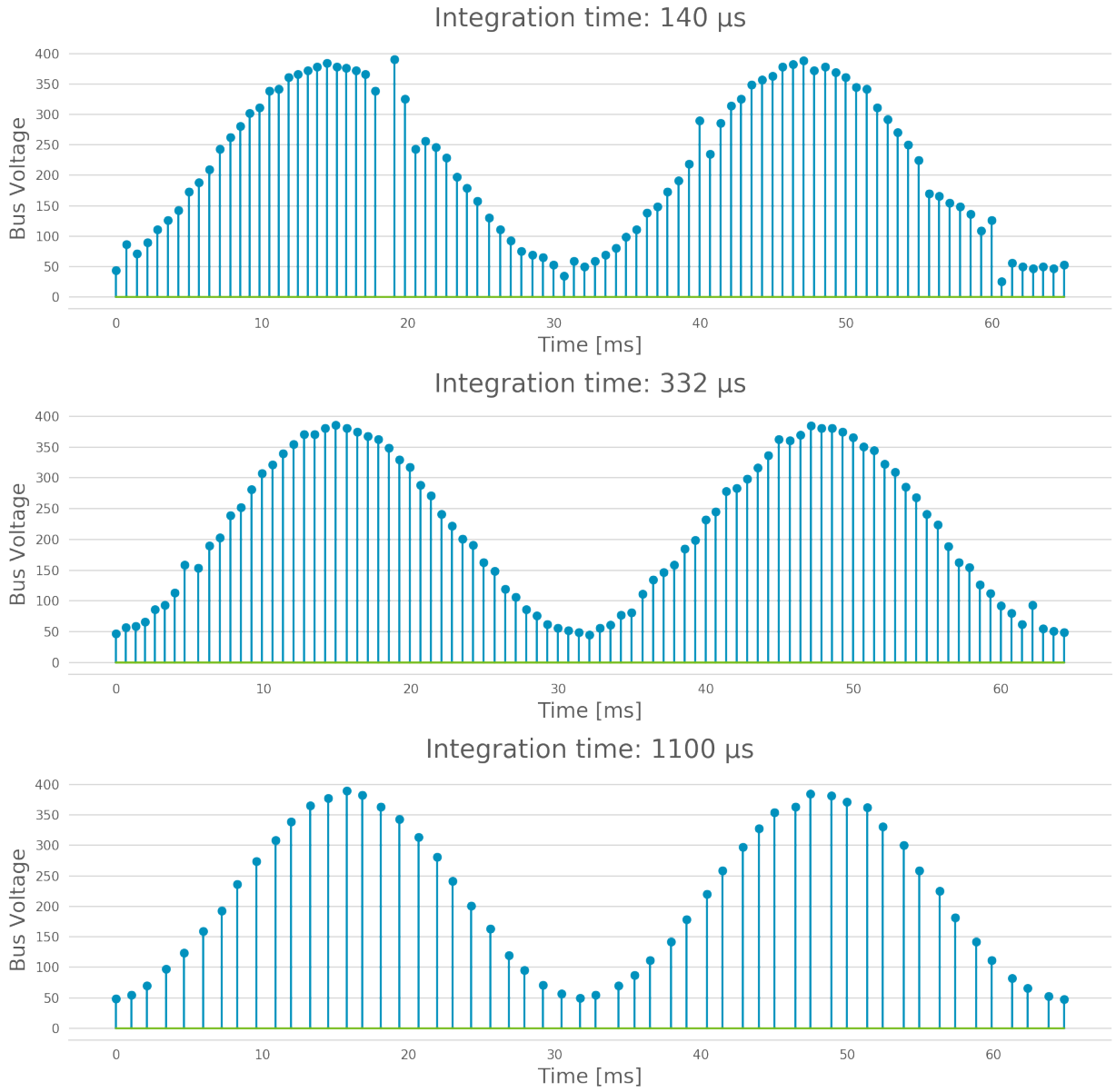


Fig. 3: Increasing the integration time increases the resilience to noise

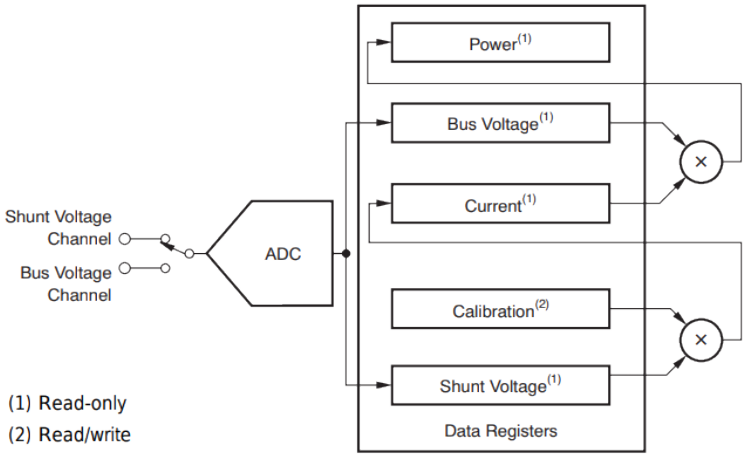


Fig. 4: Acquisition and Processing: Functional Block Diagram (source: Texas Instruments INA226)

A functional block diagram of this is also given by the datasheet:

In the end, there are therefore 3 channels (bus voltage, shunt voltage and timestamps) that are necessary to figure out the load power consumption, while the others are being provided for convenience *e.g.* in case the rest of the hardware does not have the computing power to make the computation.

Sampling Frequency Issues

It looks like the INA226-ACME-BBB setup has a bottleneck preventing the sampling frequency to go higher than ~1.4 kHz (the maximal theoretical sampling frequency is ~3.6 kHz). We know that this issue is not internal to the ADC itself (inside of the INA226) because modifying the integration time affects the output signal even when the sampling frequency is capped (as shown above) but it may come from anywhere after that.

Because of this, there is no point in using a (theoretical) sampling frequency that is larger than 1.4 kHz. But it is important to note that the ACME will still report the theoretical sampling rate (probably computed with the formula given above) through `BaylibreAcmeInstrument.sample_rate_hz` and `IIOINA226Instrument.sample_rate_hz` even if it differs from the actual sampling rate.

Note that, even though this is obvious for the theoretical sampling rate, the specific values of the bus and shunt integration times do not seem to have an influence on the measured sampling rate; only their sum matters. This further points toward a data-processing bottleneck rather than a hardware bug in the acquisition device.

The following chart compares the evolution of the measured sampling rate with the expected one as we modify it through T_{shunt}, T_{bus} and N:

Furthermore, because the transactions are done through a buffer (see next section), if the sampling frequency is too low, the connection may time-out before the buffer is full and ready to be sent. This may be fixed in an upcoming release.

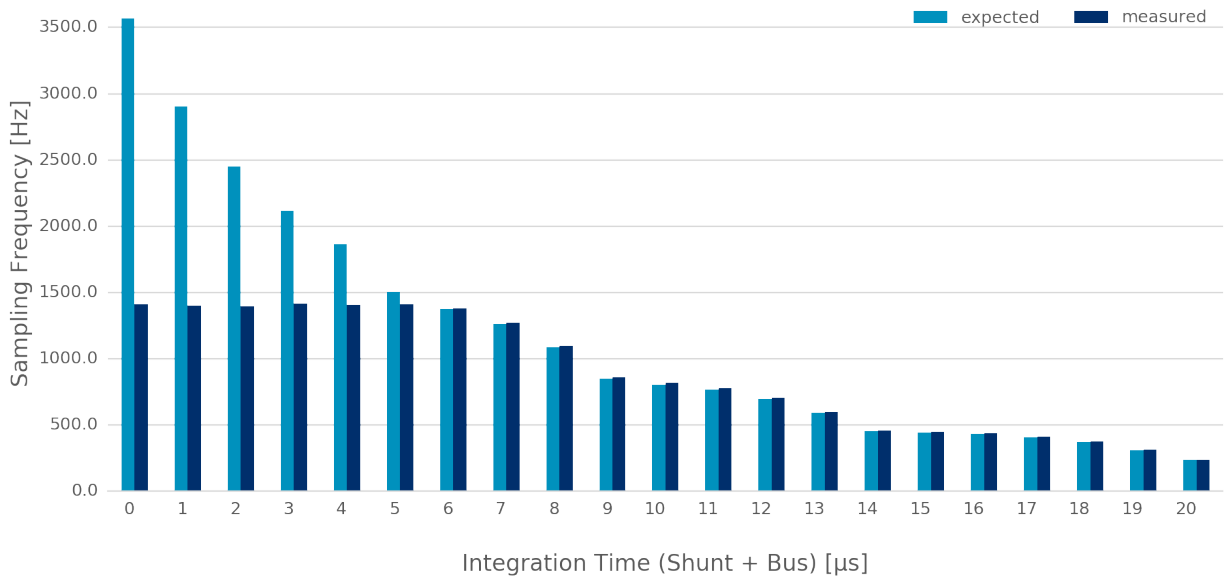


Fig. 5: Theoretical vs measured sampling rates

Buffer-based transactions

Samples made available by the INA226 are retrieved by the BBB and stored in a buffer which is sent back to the host once it is full (see `buffer_samples_count` in *`BaylibreAcmeInstrument.setup`* for setting its size). Therefore, the larger the buffer is, the longer it takes to be transmitted back but the less often it has to be transmitted. To illustrate this, consider the following graphs showing the time difference between successive samples in a retrieved signal when the size of the buffer changes:

devlib API

ACME Cape + BBB (IIO Context)

devlib provides wrapper classes for all the IIO connections to an IIO context given by libiio (the Linux IIO interface) however only the network-based one has been tested. For the other classes, please refer to the official IIO documentation for the meaning of their constructor parameters.

class devlib.instrument.baylibre_acme.**BaylibreAcmeInstrument**(*target=None*, *iio_context=None*, *use_base_iio_context=False*, *probe_names=None*)

Base class wrapper for the ACME instrument which itself is a wrapper for the IIO context base class. This class wraps around the passed `iio_context`; if `use_base_iio_context` is `True`, `iio_context` is first passed to the `iio.Context` base class (see its documentation for how this parameter is then used), else `iio_context` is expected to be a valid instance of `iio.Context`.

`probe_names` is expected to be a string or list of strings; if passed, the probes in the instance are named according to it in the order in which they are discovered (see previous comment about probe discovery and *`BaylibreAcmeInstrument.probes`*). There should be as many `probe_names` as there are probes connected to the ACME. By default, the probes keep their IIO names.

To ensure that the setup is reliable, `devlib` requires minimal versions for `iio`, the IIO drivers and the ACME BBB SD image.

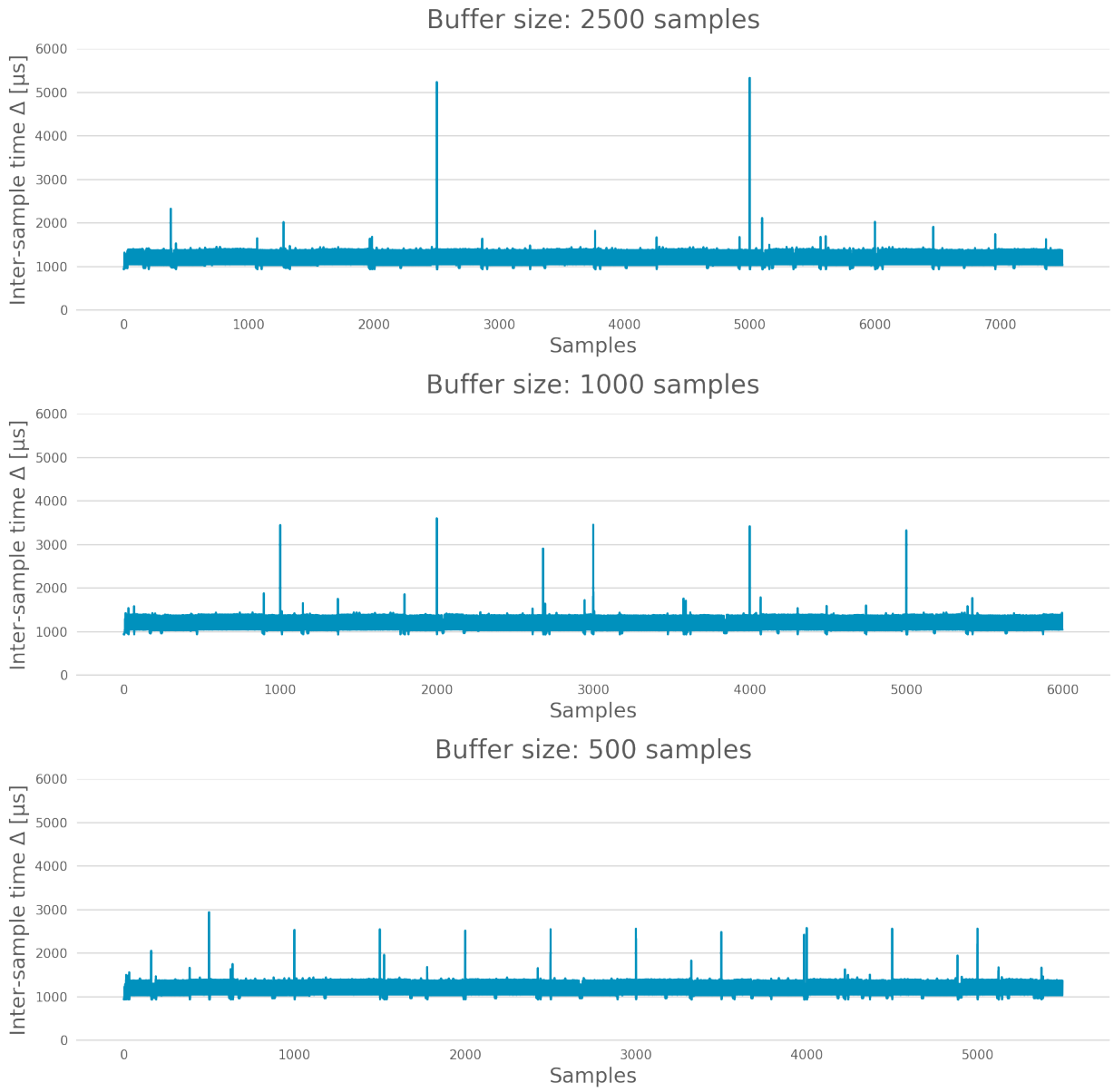


Fig. 6: Impact of the buffer size on the sampling regularity

class devlib.instrument.baylibre_acme.**BaylibreAcmeNetworkInstrument**(*target=None*, *hostname=None*, *probe_names=None*)

Child class of *BaylibreAcmeInstrument* for Ethernet-based IIO communication. The `hostname` should be the IP address or network name of the BBB. If it is `None`, the `IIOD_REMOTE` environment variable will be used as the hostname. If that environment variable is empty, the server will be discovered using ZeroConf. If that environment variable is not set, a local context is created.

class devlib.instrument.baylibre_acme.**BaylibreAcmeXMLInstrument**(*target=None*, *xmlfile=None*, *probe_names=None*)

Child class of *BaylibreAcmeInstrument* using the XML backend of the IIO library and building an IIO context from the provided `xmlfile` (a string giving the path to the file is expected).

class devlib.instrument.baylibre_acme.**BaylibreAcmeLocalInstrument**(*target=None*, *probe_names=None*)

Child class of *BaylibreAcmeInstrument* using the Local IIO backend.

BaylibreAcmeInstrument.**mode**

The collection mode for the ACME is `CONTINUOUS`.

BaylibreAcmeInstrument.**setup**(*shunt_resistor*, *integration_time_bus*, *integration_time_shunt*, *oversampling_ratio*, *buffer_samples_count=None*, *buffer_is_circular=False*, *absolute_timestamps=False*, *high_resolution=True*)

The `shunt_resistor` (R_{shunt} [$\mu\Omega$]), `integration_time_bus` (T_{bus} [s]), `integration_time_shunt` (T_{shunt} [s]) and `oversampling_ratio` (N) are copied into on-board registers inside of the INA226 to be used as described above. Please note that there exists a limited set of accepted values for these parameters; for the integration times, refer to `IIOINA226Instrument.INTEGRATION_TIMES_AVAILABLE` and for the `oversampling_ratio`, refer to `IIOINA226Instrument.OVERSAMPLING_RATIOS_AVAILABLE`. If all probes share the same value for these attributes, this class provides *BaylibreAcmeInstrument.OVERSAMPLING_RATIOS_AVAILABLE* and *BaylibreAcmeInstrument.INTEGRATION_TIMES_AVAILABLE*.

The `buffer_samples_count` is the size of the IIO buffer expressed **in samples**; this is independent of the number of active channels! By default, if `buffer_samples_count` is not passed, the IIO buffer of size *IIOINA226Instrument.sample_rate_hz* is created meaning that a buffer transfer happens roughly every second.

If `absolute_timestamps` is `False`, the first sample from the `timestamps` channel is substracted from all the following samples of this channel, effectively making its signal start at 0.

`high_resolution` is used to enable a mode where power and current are computed offline on the host machine running `devlib`: even if the user asks for power or current channels, they are not enabled in hardware (INA226) and instead the necessary voltage signal(s) are enabled to allow the computation of the desired signals using the FPU of the host (which is very likely to be much more accurate than the fixed-point 16-bit unit of the INA226).

A circular buffer can be used by setting `buffer_is_circular` to `True` (directly passed to `iio.Buffer`).

Each one of the arguments of this method can either be a single value which will be used for all probes or a list of values giving the corresponding setting for each probe (in the order of `probe_names` passed to the constructor) with the exception of `absolute_timestamps` (as all signals are resampled onto a common time signal) which, if passed as an array, will be `True` only if all of its elements are `True`.

BaylibreAcmeInstrument.**reset**(*sites=None*, *kinds=None*, *channels=None*)

BaylibreAcmeInstrument.setup() should **always** be called before calling this method so that the hardware is correctly configured. Once this method has been called, *BaylibreAcmeInstrument.setup()* can only be called again once *BaylibreAcmeInstrument.teardown()* has been called.

This method inherits from `Instrument.reset()`; call *list_channels()* for a list of available channels from a given instance.

Please note that the size of the transaction buffer is proportional to the number of active channels (for a fixed `buffer_samples_count`). Therefore, limiting the number of active channels allows to limit the required bandwidth. `high_resolution` in *BaylibreAcmeInstrument.setup()* limits the number of active channels to the minimum required.

BaylibreAcmeInstrument.**start**()

BaylibreAcmeInstrument.reset() should **always** be called before calling this method so that the right channels are active, *BaylibreAcmeInstrument.stop()* should **always** be called after calling this method and no other method of the object should be called in-between.

This method starts the sampling process of the active channels. The samples are stored but are not available until *BaylibreAcmeInstrument.stop()* has been called.

BaylibreAcmeInstrument.**stop**()

BaylibreAcmeInstrument.start() should **always** be called before calling this method so that samples are being captured.

This method stops the sampling process of the active channels and retrieves and pre-processes the samples. Once this function has been called, the samples are made available through *BaylibreAcmeInstrument.get_data()*. Note that it is safe to call *BaylibreAcmeInstrument.start()* after this method returns but this will discard the data previously acquired.

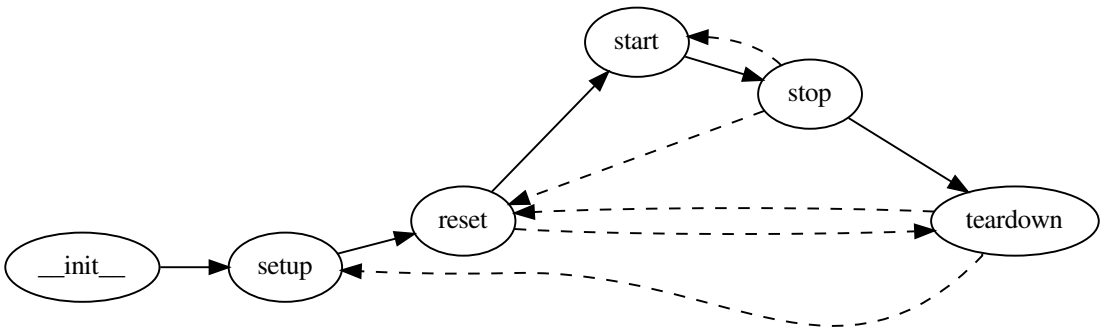
When this method returns, It is guaranteed that the content of at least one IIO buffer will have been captured.

If different sampling frequencies were used for the different probes, the signals are resampled to share the time signal with the highest sampling frequency.

BaylibreAcmeInstrument.**teardown**()

This method can be called at any point (unless otherwise specified *e.g.* *BaylibreAcmeInstrument.start()*) to deactive any active probe once *BaylibreAcmeInstrument.reset()* has been called. This method does not affect already captured samples.

The following graph gives a summary of the allowed calling sequence(s) where each edge means "can be called directly after":



BaylibreAcmeInstrument.**get_data**(*outfile=None*)

Inherited from `Instrument.get_data()`. If `outfile` is `None` (default), the samples are returned as a *pandas.DataFrame* with the channels as columns. Else, it behaves like the parent class, returning a `MeasurementCsv`.

BaylibreAcmeInstrument.**add_channel**()

Should not be used as new channels are discovered through the IIO context.

BaylibreAcmeInstrument.**list_channels**()
Inherited from Instrument.list_channels().

BaylibreAcmeInstrument.**sample_rate_hz**

BaylibreAcmeInstrument.**OVERSAMPLING_RATIOS_AVAILABLE**

BaylibreAcmeInstrument.**INTEGRATION_TIMES_AVAILABLE**
These attributes return the corresponding attributes of the probes if they all share the same value (and are therefore provided to avoid reading from a single probe and expecting the others to share this value). They should be used whenever the assumption that all probes share the same value for the accessed attribute is made. For this reason, an exception is raised if it is not the case.

If probes are active (*i.e.* *BaylibreAcmeInstrument.reset()* has been called), only these are read for the value of the attribute (as others have been tagged to be ignored). If not, all probes are used.

BaylibreAcmeInstrument.**probes**
Dictionary of *IIOINA226Instrument* instances representing the probes connected to the ACME. If provided to the constructor, the keys are the probe_names that were passed.

ACME Probes (IIO Devices)

The following class is not supposed to be instantiated by the user code: the API is provided as the ACME probes can be accessed through the *BaylibreAcmeInstrument.probes* attribute.

class devlib.instrument.baylibre_acme.**IIOINA226Instrument**(*iio_device*)
This class is a wrapper for the iio.Device class and takes a valid instance as iio_device. It is not supposed to be instantiated by the user and its partial documentation is provided for read-access only.

IIOINA226Instrument.**shunt_resistor**

IIOINA226Instrument.**sample_rate_hz**

IIOINA226Instrument.**oversampling_ratio**

IIOINA226Instrument.**integration_time_shunt**

IIOINA226Instrument.**integration_time_bus**

IIOINA226Instrument.**OVERSAMPLING_RATIOS_AVAILABLE**

IIOINA226Instrument.**INTEGRATION_TIMES_AVAILABLE**
These attributes are provided *for reference* and should not be assigned to but can be used to make the user code more readable, if needed. Please note that, as reading these attributes reads the underlying value from the hardware, they should not be read when the ACME is active *i.e* when *BaylibreAcmeInstrument.setup()* has been called without calling *BaylibreAcmeInstrument.teardown()*.

Examples

The following example shows a basic use of an ACME at IP address ACME_IP_ADDR with 2 probes connected, capturing all the channels during (roughly) 10 seconds at a sampling rate of 613 Hz and outputing the measurements to the CSV file acme.csv:

```
import time
import devlib

acme = devlib.BaylibreAcmeNetworkInstrument(hostname=ACME_IP_ADDR,
                                            probe_names=['battery', 'usb'])
```

(continues on next page)

(continued from previous page)

```
int_times = acme.INTEGRATION_TIMES_AVAILABLE
ratios    = acme.OVERSAMPLING_RATIOS_AVAILABLE

acme.setup(shunt_resistor=20000,
           integration_time_bus=int_times[1],
           integration_time_shunt=int_times[1],
           oversampling_ratio=ratios[1])

acme.reset()
acme.start()
time.sleep(10)
acme.stop()
acme.get_data('acme.csv')
acme.teardown()
```

It is common to have different resistances for different probe shunt resistors. Furthermore, we may want to have different sampling frequencies for different probes (*e.g.* if it is known that the USB voltage changes rather slowly). Finally, it is possible to set the integration times for the bus and shunt voltages of a same probe to different values. The following call to *BaylibreAcmeInstrument.setup()* illustrates these:

```
acme.setup(shunt_resistor=[20000, 10000],
           integration_time_bus=[int_times[2], int_times[3]],
           integration_time_shunt=[int_times[3], int_times[4]],
           oversampling_ratio=[ratios[0], ratios[1]])

for n, p in acme.probes.iteritems():
   print('{}:'.format(n))
   print('    T_bus = {} s'.format(p.integration_time_bus))
   print('    T_shn = {} s'.format(p.integration_time_shunt))
   print('    N     = {}'.format(p.oversampling_ratio))
   print('    freq  = {} Hz'.format(p.sample_rate_hz))

# Output:
#
#   battery:
#       T_bus = 0.000332 s
#       T_shn = 0.000588 s
#       N     = 1
#       freq  = 1087 Hz
#   usb:
#       T_bus = 0.000588 s
#       T_shn = 0.0011 s
#       N     = 4
#       freq  = 148 Hz
```

Please keep in mind that calling `acme.get_data('acme.csv')` after capturing samples with this setup will output signals with the same sampling frequency (the highest one among the sampling frequencies) as the signals are resampled to output a single time signal.

CHAPTER

FIVE

COLLECTORS

The `Collector` API provide a consistent way of collecting arbitrary data from a target. Data is collected via an instance of a class derived from `CollectorBase`.

5.1 Example

The following example shows how to use a collector to read the logcat output from an Android target.

```
# import and instantiate the Target and the collector
# (note: this assumes exactly one android target connected
#  to the host machine).
In [1]: from devlib import AndroidTarget, LogcatCollector

In [2]: t = AndroidTarget()

# Set up the collector on the Target.

In [3]: collector = LogcatCollector(t)

# Configure the output file path for the collector to use.
In [4]: collector.set_output('adb_log.txt')

# Reset the Collector to preform any required configuration or preparation.
In [5]: collector.reset()

# Start Collecting
In [6]: collector.start()

# Wait for some output to be generated
In [7]: sleep(10)

# Stop Collecting
In [8]: collector.stop()

# Retrieved the collected data
In [9]: output = collector.get_data()

# Display the returned ``CollectorOutput`` Object.
In [10]: output
Out[10]: [<adb_log.txt (file)>]
```

(continues on next page)

(continued from previous page)

```
In [11] log_file = output[0]

# Get the path kind of the the returned CollectorOutputEntry.
In [12]: log_file.path_kind
Out[12]: 'file'

# Get the path of the returned CollectorOutputEntry.
In [13]: log_file.path
Out[13]: 'adb_log.txt'

# Find the full path to the log file.
In [14]: os.path.join(os.getcwd(), logfile)
Out[14]: '/tmp/adb_log.txt'
```

5.2 API

5.2.1 CollectorBase

class devlib.collector.**CollectorBase**(*target*, ***kwargs*)

A `CollectorBase` is the the base class and API that should be implemented to allowing collecting various data from a traget e.g. traces, logs etc.

`Collector.`**`setup`**(**args*, ***kwargs*)

This will set up the collector on the target. Parameters this method takes are particular to subclasses (see documentation for specific collectors below). What actions are performed by this method are also collector-specific. Usually these will be things like installing executables, starting services, deploying assets, etc. Typically, this method needs to be invoked at most once per reboot of the target (unless `teardown()` has been called), but see documentation for the collector you're interested in.

`CollectorBase.`**`reset`**()

This can be used to configure a collector for collection. This must be invoked before `start()` is called to begin collection.

`CollectorBase.`**`start`**()

Starts collecting from the target.

`CollectorBase.`**`stop`**()

Stops collecting from target. Must be called after *start()*.

`CollectorBase.`**`set_output`**(*output_path*)

Configure the output path for the particular collector. This will be either a directory or file path which will be used when storing the data. Please see the individual Collector documentation for more information.

`CollectorBase.`**`get_data`**()

The collected data will be return via the previously specified output_path. This method will return a `CollectorOutput` object which is a subclassed list object containing individual `CollectorOutputEntry` objects with details about the individual output entry.

5.2.2 CollectorOutputEntry

This object is designed to allow for the output of a collector to be processed generically. The object will behave as a regular string containing the path to underlying output path and can be used directly in `os.path` operations.

`CollectorOutputEntry.`**`path`**
The file path for the corresponding output item.

`CollectorOutputEntry.`**`path_kind`**
The type of output the is specified in the `path` attribute. Current valid kinds are: `file` and `directory`.

`CollectorOutputEntry.`**`__init__`**(*path*, *path_kind*)
Initialises a `CollectorOutputEntry` object with the desired file path and kind of file path specified.

5.3 Available Collectors

This section lists collectors that are currently part of devlib.

Todo: Add collectors

CHAPTER

SIX

DERIVED MEASUREMENTS

The `DerivedMeasurements` API provides a consistent way of performing post processing on a provided `MeasurementCsv` file.

6.1 Example

The following example shows how to use an implementation of a `DerivedMeasurement` to obtain a list of calculated `DerivedMetric`'s.

```
# Import the relevant derived measurement module
# in this example the derived energy module is used.
In [1]: from devlib import DerivedEnergyMeasurements

# Obtain a MeasurementCsv file from an instrument or create from
# existing .csv file. In this example an existing csv file is used which was
# created with a sampling rate of 100Hz
In [2]: from devlib import MeasurementsCsv
In [3]: measurement_csv = MeasurementsCsv('/example/measurements.csv', sample_rate_
↪hz=100)

# Process the file and obtain a list of the derived measurements
In [4]: derived_measurements = DerivedEnergyMeasurements.process(measurement_csv)

In [5]: derived_measurements
Out[5]: [device_energy: 239.1854075 joules, device_power: 5.5494089227 watts]
```

6.2 API

6.2.1 Derived Measurements

class devlib.derived.**DerivedMeasurements**

The `DerivedMeasurements` class provides an API for post-processing instrument output offline (i.e. without a connection to the target device) to generate additional metrics.

`DerivedMeasurements.`**process**(*measurement_csv*)

Process a `MeasurementsCsv`, returning a list of *`DerivedMetric`* and/or `MeasurementsCsv` objects that have been derived from the input. The exact nature and ordering of the list members is specific to individual 'class'*DerivedMeasurements* implementations.

DerivedMeasurements.**process_raw**(**args*)
Process raw output from an instrument, returning a list *DerivedMetric* and/or MeasurementsCsv objects that have been derived from the input. The exact nature and ordering of the list members is specific to individual 'class'*DerivedMeasurements* implementations.

The arguments to this method should be paths to raw output files generated by an instrument. The number and order of expected arguments is specific to particular implementations.

6.2.2 Derived Metric

class devlib.derived.**DerivedMetric**
Represents a metric derived from previously collected Measurement``s. Unlike, a ``Measurement, this was not measured directly from the target.

DerivedMetric.**name**
The name of the derived metric. This uniquely defines a metric – two DerivedMetric objects with the same name represent to instances of the same metric (e.g. computed from two different inputs).

DerivedMetric.**value**
The numeric value of the metric that has been computed for a particular input.

DerivedMetric.**measurement_type**
The MeasurementType of the metric. This indicates which conceptual category the metric falls into, its units, and conversions to other measurement types.

DerivedMetric.**units**
The units in which the metric's value is expressed.

6.3 Available Derived Measurements

Note: If a method of the API is not documented for a particular implementation, that means that it s not overridden by that implementation. It is still safe to call it – an empty list will be returned.

6.3.1 Energy

class devlib.derived.energy.**DerivedEnergyMeasurements**
The DerivedEnergyMeasurements class is used to calculate average power and cumulative energy for each site if the required data is present.

The calculation of cumulative energy can occur in 3 ways. If a site contains energy results, the first and last measurements are extracted and the delta calculated. If not, a timestamp channel will be used to calculate the energy from the power channel, failing back to using the sample rate attribute of the MeasurementCsv file if timestamps are not available. If neither timestamps or a sample rate are available then an error will be raised.

DerivedEnergyMeasurements.**process**(*measurement_csv*)
This will return total cumulative energy for each energy channel, and the average power for each power channel in the input CSV. The output will contain all energy metrics followed by power metrics. The ordering of both will match the ordering of channels in the input. The metrics will by named based on the sites of the corresponding channels according to the following patters: "<site>_total_energy" and "<site>_average_power".

6.3.2 FPS / Rendering

class devlib.derived.fps.**DerivedGfxInfoStats**(*drop_threshold=5*, *suffix='-fps'*, *filename=None*, *outdir=None*)

Produces FPS (frames-per-second) and other derived statistics from `GfxInfoFramesInstrument` output. This takes several optional parameters in creation:

Parameters

- **drop_threshold** – FPS in an application, such as a game, which this processor is primarily targeted at, cannot reasonably drop to a very low value. This is specified to this threshold. If an FPS for a frame is computed to be lower than this threshold, it will be dropped on the assumption that frame rendering was suspended by the system (e.g. when idling), or there was some sort of error, and therefore this should be used in performance calculations. defaults to `5`.
- **suffix** – The name of the generated per-frame FPS csv file will be derived from the input frames csv file by appending this suffix. This cannot be specified at the same time as a `filename`.
- **filename** – As an alternative to the suffix, a complete file name for FPS csv can be specified. This cannot be used at the same time as the `suffix`.
- **outdir** – By default, the FPS csv file will be placed in the same directory as the input frames csv file. This can be changed by specifying an alternate directory here

Warning: Specifying both `filename` and `oudir` will mean that exactly the same file will be used for FPS output on each invocation of `process()` (even for different inputs) resulting in previous results being overwritten.

DerivedGfxInfoStats.**process**(*measurement_csv*)

Process the fames csv generated by `GfxInfoFramesInstrument` and returns a list containing exactly three entries: `DerivedMetrics` `fps` and `total_frames`, followed by a `MeasurentCsv` containing per-frame FPSs values.

DerivedGfxInfoStats.**process_raw**(*gfxinfo_frame_raw_file*)

As input, this takes a single argument, which should be the path to the raw output file of `GfxInfoFramesInstrument`. The returns stats accumulated by gfxinfo. At the time of writing, the stats (in order) are: `janks`, `janks_pc` (percentage of all frames), `render_time_50th_ptile` (50th percentile, or median, for time to render a frame), `render_time_90th_ptile`, `render_time_95th_ptile`, `render_time_99th_ptile`, `missed_vsync`, `hight_input_latency`, `slow_ui_thread`, `slow_bitmap_uploads`, `slow_issue_draw_commands`. Please see the gfxinfo documentation for details.

class devlib.derived.fps.**DerivedSurfaceFlingerStats**(*drop_threshold=5*, *suffix='-fps'*, *filename=None*, *outdir=None*)

Produces FPS (frames-per-second) and other derived statistics from `SurfaceFlingerFramesInstrument` output. This takes several optional parameters in creation:

Parameters

- **drop_threshold** – FPS in an application, such as a game, which this processor is primarily targeted at, cannot reasonably drop to a very low value. This is specified to this threshold. If an FPS for a frame is computed to be lower than this threshold, it will be dropped on the assumption that frame rendering was suspended by the system (e.g. when idling), or there was some sort of error, and therefore this should be used in performance calculations. defaults to `5`.

- **suffix** – The name of the generated per-frame FPS csv file will be derived from the input frames csv file by appending this suffix. This cannot be specified at the same time as a `filename`.
- **filename** – As an alternative to the suffix, a complete file name for FPS csv can be specified. This cannot be used at the same time as the `suffix`.
- **outdir** – By default, the FPS csv file will be placed in the same directory as the input frames csv file. This can be changed by specifying an alternate directory here

Warning: Specifying both `filename` and `oudir` will mean that exactly the same file will be used for FPS output on each invocation of `process()` (even for different inputs) resulting in previous results being overwritten.

`DerivedSurfaceFlingerStats.`**process**(*measurement_csv*)

Process the fames csv generated by `SurfaceFlingerFramesInstrument` and returns a list containing exactly three entries: `DerivedMetrics` `fps` and `total_frames`, followed by a `MeasurentCsv` containing per-frame FPSs values, followed by `janks` `janks_pc`, and `missed_vsync` metrics.

CHAPTER

SEVEN

PLATFORM

*Platform*s describe the system underlying the OS. They encapsulate hardware- and firmware-specific details. In most cases, the generic *Platform* class, which gets used if a platform is not explicitly specified on *Target* creation, will be sufficient. It will automatically query as much platform information (such CPU topology, hardware model, etc) if it was not specified explicitly by the user.

class devlib.platform.**Platform**(*name=None*, *core_names=None*, *core_clusters=None*, *big_core=None*, *model=None*, *modules=None*)

Parameters

- **name** – A user-friendly identifier for the platform.
- **core_names** – A list of CPU core names in the order they appear registered with the OS. If they are not specified, they will be queried at run time.
- **core_clusters** – A list with cluster ids of each core (starting with 0). If this is not specified, clusters will be inferred from core names (cores with the same name are assumed to be in a cluster).
- **big_core** – The name of the big core in a big.LITTLE system. If this is not specified it will be inferred (on systems with exactly two clusters).
- **model** – Model name of the hardware system. If this is not specified it will be queried at run time.
- **modules** – Modules with additional functionality supported by the platform (e.g. for handling flashing, rebooting, etc). These would be added to the Target's modules. (See *Modules*).

7.1 Versatile Express

The generic platform may be extended to support hardware- or infrastructure-specific functionality. Platforms exist for ARM VersatileExpress-based Juno and TC2 development boards. In addition to the standard *Platform* parameters above, these platforms support additional configuration:

class devlib.platform.arm.**VersatileExpressPlatform**

Normally, this would be instantiated via one of its derived classes (Juno or TC2) that set appropriate defaults for some of the parameters.

Parameters

- **serial_port** – Identifies the serial port (usual a /dev node) on which the device is connected.

- **baudrate** – Baud rate for the serial connection. This defaults to `115200` for `Juno` and `38400` for `TC2`.
- **vemsd_mount** – Mount point for the VEMSD (Versatile Express MicroSD card that is used for board configuration files and firmware images). This defaults to `"/media/JUNO"` for `Juno` and `"/media/VEMSD"` for `TC2`, though you would most likely need to change this for your setup as it would depend both on the file system label on the MicroSD card, and on how the card was mounted on the host system.
- **hard_reset_method** – Specifies the method for hard-resetting the devices (e.g. if it becomes unresponsive and normal reboot method doesn't not work). Currently supported methods are:

  dtr reboot by toggling DTR line on the serial connection (this is enabled via a DIP switch on the board).

  reboottxt reboot by writing a filed called `reboot.txt` to the root of the VEMSD mount (this is enabled via board configuration file).

  This defaults to `dtr` for `Juno` and `reboottxt` for `TC2`.
- **bootloader** – Specifies the bootloader configuration used by the board. The following values are currently supported:

  uefi Boot via UEFI menu, by selecting the entry specified by `uefi_entry` parameter. If this entry does not exist, it will be automatically created based on values provided for `image`, `initrd`, `fdt`, and `bootargs` parameters.

  uefi-shell Boot by going via the UEFI shell.

  u-boot Boot using Das U-Boot.

  bootmon Boot directly via Versatile Express Bootmon using the values provided for `image`, `initrd`, `fdt`, and `bootargs` parameters.

  This defaults to `u-boot` for `Juno` and `bootmon` for `TC2`.
- **flash_method** – Specifies how the device is flashed. Currently, only `"vemsd"` method is supported, which flashes by writing firmware images to an appropriate location on the VEMSD.
- **image** – Specfies the kernel image name for `uefi` or `bootmon` boot.
- **fdt** – Specifies the device tree blob for `uefi` or `bootmon` boot.
- **initrd** – Specifies the ramdisk image for `uefi` or `bootmon` boot.
- **bootargs** – Specifies the boot arguments that will be pass to the kernel by the bootloader.
- **uefi_entry** – Then name of the UEFI entry to be used/created by `uefi` bootloader.
- **ready_timeout** – Timeout, in seconds, for the time it takes the platform to become ready to accept connections. Note: this does not mean that the system is fully booted; just that the services needed to establish a connection (e.g. sshd or adbd) are up.

7.2 Gem5 Simulation Platform

By initialising a Gem5SimulationPlatform, devlib will start a gem5 simulation (based upon the arguments the user provided) and then connect to it using *Gem5Connection*. Using the methods discussed above, some methods of the *Target* will be altered slightly to better suit gem5.

class devlib.platform.gem5.**Gem5SimulationPlatform**(*name*, *host_output_dir*, *gem5_bin*, *gem5_args*, *gem5_virtio*, *gem5_telnet_port=None*)

During initialisation the gem5 simulation will be kicked off (based upon the arguments provided by the user) and the telnet port used by the gem5 simulation will be intercepted and stored for use by the *Gem5Connection*.

Parameters

- **name** – Platform name
- **host_output_dir** – Path on the host where the gem5 outputs will be placed (e.g. stats file)
- **gem5_bin** – gem5 binary
- **gem5_args** – Arguments to be passed onto gem5 such as config file etc.
- **gem5_virtio** – Arguments to be passed onto gem5 in terms of the virtIO device used to transfer files between the host and the gem5 simulated system.
- **gem5_telnet_port** – Not yet in use as it would be used in future implementations of devlib in which the user could use the platform to pick up an existing and running simulation.

Gem5SimulationPlatform.**init_target_connection**([*target*])

Based upon the OS defined in the *Target*, the type of *Gem5Connection* will be set (*AndroidGem5Connection* or *AndroidGem5Connection*).

Gem5SimulationPlatform.**update_from_target**([*target*])

This method provides specific setup procedures for a gem5 simulation. First of all, the m5 binary will be installed on the guest (if it is not present). Secondly, three methods in the *Target* will be monkey-patched:

- **reboot**: this is not supported in gem5
- **reset**: this is not supported in gem5
- **capture_screen**: gem5 might already have screencaps so the monkey-patched method will first try to transfer the existing screencaps. In case that does not work, it will fall back to the original *Target* implementation of `capture_screen()`.

Finally, it will call the parent implementation of *update_from_target()*.

Gem5SimulationPlatform.**setup**([*target*])

The m5 binary be installed, if not yet installed on the gem5 simulated system. It will also resize the gem5 shell, to avoid line wrapping issues.

CHAPTER

EIGHT

CONNECTION

A `Connection` abstracts an actual physical connection to a device. The first connection is created when `Target.connect()` method is called. If a *Target* is used in a multi-threaded environment, it will maintain a connection for each thread in which it is invoked. This allows the same target object to be used in parallel in multiple threads.

`Connection`s will be automatically created and managed by *Target*s, so there is usually no reason to create one manually. Instead, configuration for a `Connection` is passed as *connection_settings* parameter when creating a *Target*. The connection to be used target is also specified on instantiation by *conn_cls* parameter, though all concrete *Target* implementations will set an appropriate default, so there is typically no need to specify this explicitly.

`Connection` classes are not a part of an inheritance hierarchy, i.e. they do not derive from a common base. Instead, a `Connection` is any class that implements the following methods.

push(*self*, *sources*, *dest*, *timeout=None*)

Transfer a list of files from the host machine to the connected device.

Parameters

- **sources** – list of paths on the host
- **dest** – path to the file or folder on the connected device.
- **timeout** – timeout (in seconds) for the transfer of each file; if the transfer does not complete within this period, an exception will be raised.

pull(*self*, *sources*, *dest*, *timeout=None*)

Transfer a list of files from the connected device to the host machine.

Parameters

- **sources** – list of paths on the connected device.
- **dest** – path to the file or folder on the host
- **timeout** – timeout (in seconds) for the transfer for each file; if the transfer does not complete within this period, an exception will be raised.

execute(*self*, *command*, *timeout=None*, *check_exit_code=False*, *as_root=False*, *strip_colors=True*, *will_succeed=False*)

Execute the specified command on the connected device and return its output.

Parameters

- **command** – The command to be executed.
- **timeout** – Timeout (in seconds) for the execution of the command. If specified, an exception will be raised if execution does not complete with the specified period.
- **check_exit_code** – If `True` the exit code (on connected device) from execution of the command will be checked, and an exception will be raised if it is not `0`.

- **as_root** – The command will be executed as root. This will fail on unrooted connected devices.
- **strip_colours** – The command output will have colour encodings and most ANSI escape sequences striped out before returning.
- **will_succeed** – The command is assumed to always succeed, unless there is an issue in the environment like the loss of network connectivity. That will make the method always raise an instance of a subclass of `DevlibTransientError` when the command fails, instead of a `DevlibStableError`.

background(*self*, *command*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, *as_root=False*)

Execute the command on the connected device, invoking it via subprocess on the host. This will return `subprocess.Popen` instance for the command.

Parameters

- **command** – The command to be executed.
- **stdout** – By default, standard output will be piped from the subprocess; this may be used to redirect it to an alternative file handle.
- **stderr** – By default, standard error will be piped from the subprocess; this may be used to redirect it to an alternative file handle.
- **as_root** – The command will be executed as root. This will fail on unrooted connected devices.

Note: This **will block the connection** until the command completes.

Note: The above methods are directly wrapped by *Target* methods, however note that some of the defaults are different.

cancel_running_command(*self*)

Cancel a running command (previously started with *background()*) and free up the connection. It is valid to call this if the command has already terminated (or if no command was issued), in which case this is a no-op.

close(*self*)

Close the connection to the device. The `Connection` object should not be used after this method is called. There is no way to reopen a previously closed connection, a new connection object should be created instead.

Note: There is no `open()` method, as the connection is assumed to be opened on instantiation.

8.1 Connection Types

class devlib.utils.android.**AdbConnection**(*device=None*, *timeout=None*, *adb_server=None*, *adb_as_root=False*, *connection_attempts=MAX_ATTEMPTS*, *poll_transfers=False*, *start_transfer_poll_delay=30*, *total_transfer_timeout=3600*, *transfer_poll_period=30*)

A connection to an android device via `adb` (Android Debug Bridge). `adb` is part of the Android SDK (though stand-alone versions are also available).

Parameters

- **device** – The name of the adb device. This is usually a unique hex string for USB-connected devices, or an ip address/port combination. To see connected devices, you can run `adb devices` on the host.
- **timeout** – Connection timeout in seconds. If a connection to the device is not established within this period, `HostError` is raised.
- **adb_server** – Allows specifying the address of the adb server to use.
- **adb_as_root** – Specify whether the adb server should be restarted in root mode.
- **connection_attempts** – Specify how many connection attempts, 10 seconds apart, should be attempted to connect to the device. Defaults to 5.
- **poll_transfers** – Specify whether file transfers should be polled. Polling monitors the progress of file transfers and periodically checks whether they have stalled, attempting to cancel the transfers prematurely if so.
- **start_transfer_poll_delay** – If transfers are polled, specify the length of time after a transfer has started before polling should start.
- **total_transfer_timeout** – If transfers are polled, specify the total amount of time to elapse before the transfer is cancelled, regardless of its activity.
- **transfer_poll_period** – If transfers are polled, specify the period at which the transfers are sampled for activity. Too small values may cause the destination size to appear the same over one or more sample periods, causing improper transfer cancellation.

class devlib.utils.ssh.**SshConnection**(*host*, *username*, *password=None*, *keyfile=None*, *port=22*, *timeout=None*, *platform=None*, *sudo_cmd='sudo -- sh -c {}'*, *strict_host_check=True*, *use_scp=False*, *poll_transfers=False*, *start_transfer_poll_delay=30*, *total_transfer_timeout=3600*, *transfer_poll_period=30*)

A connection to a device on the network over SSH.

Parameters

- **host** – SSH host to which to connect
- **username** – username for SSH login
- **password** – password for the SSH connection

Note: To connect to a system without a password this parameter should be set to an empty string otherwise ssh key authentication will be attempted.

Note: In order to user password-based authentication, `sshpass` utility must be installed on the system.

- **keyfile** – Path to the SSH private key to be used for the connection.

Note: `keyfile` and `password` can't be specified at the same time.

- **port** – TCP port on which SSH server is listening on the remote device. Omit to use the default port.

- **timeout** – Timeout for the connection in seconds. If a connection cannot be established within this time, an error will be raised.
- **platform** – Specify the platform to be used. The generic *Platform* class is used by default.
- **sudo_cmd** – Specify the format of the command used to grant sudo access.
- **strict_host_check** – Specify the ssh connection parameter `StrictHostKeyChecking`,
- **use_scp** – Use SCP for file transfers, defaults to SFTP.
- **poll_transfers** – Specify whether file transfers should be polled. Polling monitors the progress of file transfers and periodically checks whether they have stalled, attempting to cancel the transfers prematurely if so.
- **start_transfer_poll_delay** – If transfers are polled, specify the length of time after a transfer has started before polling should start.
- **total_transfer_timeout** – If transfers are polled, specify the total amount of time to elapse before the transfer is cancelled, regardless of its activity.
- **transfer_poll_period** – If transfers are polled, specify the period at which the transfers are sampled for activity. Too small values may cause the destination size to appear the same over one or more sample periods, causing improper transfer cancellation.

class devlib.utils.ssh.**TelnetConnection**(*host*, *username*, *password=None*, *port=None*, *timeout=None*, *password_prompt=None*, *original_prompt=None*)

A connection to a device on the network over Telenet.

Note: Since Telenet protocol is does not support file transfer, scp is used for that purpose.

Parameters

- **host** – SSH host to which to connect
- **username** – username for SSH login
- **password** – password for the SSH connection

  Note: In order to user password-based authentication, `sshpass` utility must be installed on the system.

- **port** – TCP port on which SSH server is listening on the remote device. Omit to use the default port.
- **timeout** – Timeout for the connection in seconds. If a connection cannot be established within this time, an error will be raised.
- **password_prompt** – A string with the password prompt used by `sshpass`. Set this if your version of `sshpass` uses something other than `"[sudo] password"`.
- **original_prompt** – A regex for the shell prompted presented in the Telenet connection (the prompt will be reset to a randomly-generated pattern for the duration of the connection to reduce the possibility of clashes). This parameter is ignored for SSH connections.

class devlib.host.**LocalConnection**(*keep_password=True*, *unrooted=False*, *password=None*)

A connection to the local host allowing it to be treated as a Target.

Parameters

- **keep_password** – If this is `True` (the default) user's password will be cached in memory after it is first requested.
- **unrooted** – If set to `True`, the platform will be assumed to be unrooted without testing for root. This is useful to avoid blocking on password request in scripts.
- **password** – Specify password on connection creation rather than prompting for it.

class devlib.utils.ssh.**Gem5Connection**(*platform*, *host=None*, *username=None*, *password=None*, *timeout=None*, *password_prompt=None*, *original_prompt=None*)

A connection to a gem5 simulation using a local Telnet connection.

Note: Some of the following input parameters are optional and will be ignored during initialisation. They were kept to keep the analogy with a *TelnetConnection* (i.e. `host`, `username`, `password`, `port`, `password_prompt` and `original_promp`)

Parameters

- **host** – Host on which the gem5 simulation is running

  Note: Even though the input parameter for the `host` will be ignored, the gem5 simulation needs to be on the same host the user is currently on, so if the host given as input parameter is not the same as the actual host, a `TargetStableError` will be raised to prevent confusion.

- **username** – Username in the simulated system
- **password** – No password required in gem5 so does not need to be set
- **port** – Telnet port to connect to gem5. This does not need to be set at initialisation as this will either be determined by the `Gem5SimulationPlatform` or can be set using the `connect_gem5()` method
- **timeout** – Timeout for the connection in seconds. Gem5 has high latencies so unless the timeout given by the user via this input parameter is higher than the default one (3600 seconds), this input parameter will be ignored.
- **password_prompt** – A string with password prompt
- **original_prompt** – A regex for the shell prompt

There are two classes that inherit from *Gem5Connection*: *AndroidGem5Connection* and *LinuxGem5Connection*. They inherit *almost* all methods from the parent class, without altering them. The only methods discussed below are those that will be overwritten by the *LinuxGem5Connection* and *AndroidGem5Connection* respectively.

class devlib.utils.ssh.**LinuxGem5Connection**

A connection to a gem5 simulation that emulates a Linux system.

_login_to_device(*self*)

Login to the gem5 simulated system.

class devlib.utils.ssh.**AndroidGem5Connection**

A connection to a gem5 simulation that emulates an Android system.

_wait_for_boot(*self*)

Wait for the gem5 simulated system to have booted and finished the booting animation.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W