Antmicro

Protoplaster - docs example

2025-10-31

CONTENTS

CHAPTER

ONE

INTRODUCTION

This documentation serves as an example of how individual projects documentation can look like.

The second chapter contains reference of remote API when running Protoplaster in server mode.

The third chapter contains information from the README file.

The last chapter is generated from the sample `test.yml` file which can be found in the README. Its purpose is to demonstrate the documentation generated to describe test procedures used in a project.

CHAPTER

TWO

PROTOPLASTER

An automated framework for platform testing (Hardware and BSPs).

Currently includes tests for:

- I2C
- GPIO
- Camera
- FPGA

2.1 Installation

```
pip install git+https://github.com/antmicro/protoplaster.git
```

2.2 Usage

```
usage: protoplaster [-h] [-d TEST_DIR] [-r REPORTS_DIR] [-a ARTIFACTS_DIR]
                    [-t TEST_FILE] [-g GROUP] [-s TEST_SUITE] [--list-groups]
                    [--list-test-suites] [--list-tests] [-o OUTPUT]
                    [--csv CSV] [--csv-columns CSV_COLUMNS] [--generate-docs]
                    [-c CUSTOM_TESTS] [--report-output REPORT_OUTPUT]
                    [--system-report-config SYSTEM_REPORT_CONFIG] [--sudo]
                    [--server]

options:
  -h, --help            show this help message and exit
  -d, --test-dir TEST_DIR
                        Path to the test directory
  -r, --reports-dir REPORTS_DIR
                        Path to the reports directory
  -a, --artifacts-dir ARTIFACTS_DIR
                        Path to the test artifacts directory
  -t, --test-file TEST_FILE
                        Path to the yaml test description in the test
                        directory
  -g, --group GROUP     Group to execute [deprecated]
```

(continues on next page)

(continued from previous page)

```
  -s, --test-suite TEST_SUITE
                        Test suite to execute
  --list-groups         List possible groups to execute [deprecated]
  --list-test-suites    List possible test suites to execute
  --list-tests          List all defined tests
  -o, --output OUTPUT   A junit-xml style report of the tests results
  --csv CSV             Generate a CSV report of the tests results
  --csv-columns CSV_COLUMNS
                        Comma-separated list of columns to be included in
                        generated CSV
  --generate-docs       Generate documentation
  -c, --custom-tests CUSTOM_TESTS
                        Path to the custom tests sources
  -l, --log             Append test results to a log file
  --report-output REPORT_OUTPUT
                        Proplaster report archive
  --system-report-config SYSTEM_REPORT_CONFIG
                        Path to the system report yaml config file
  --sudo                Run as sudo
  --server              Run in server mode
```

Protoplaster expects a yaml file describing tests as an input. The yaml file should have a structure specified as follows:

```yaml
includes:
  - addition.yml        # Import additional definitions from external file

tests:
  base:                 # Test name
    i2c:                # A module specifier
    - bus: 0            # An interface specifier
      devices:          # Multiple instances of devices can be defined in one␣
↪module
      - name: "Sensor name"
        address: 0x3c   # The given device parameters determine which tests will␣
↪be run for the module
    - bus: 0
      devices:
      - name: "I2C-bus multiplexer"
        address: 0x70
    camera:
    - device: "/dev/video0"
      camera_name: "vivid"
      driver_name: "vivid"
    - device: "/dev/video2"
      camera_name: "vivid"
      driver_name: "vivid"
      save_file: "frame.raw"
  additional:
```

(continues on next page)

(continued from previous page)

```
    gpio:
    - number: 20
      value: 1

metadata:               # Additional metadata to be generated on tested device
  uname:                # Metadata name
    run: uname -r       # Command to run

test-suites:
  basic:                # Test suite name
    tests:              # Tests to include
      - base
  full:
    tests:
      - basic           # Test suites can include other test suites
      - additional
    metadata:           # Metadata to generate for this test
      - uname
```

2.2.1 Test suites

In the YAML file, you can define different groups of tests in the `test-suites` section to run them for different use cases. In the YAML file example, there are two suites defined: `basic` and `full`. Protoplaster, when run without a defined test suite, will execute all tests defined in given file. When the test suite is specified with the parameter `-s` or `--test-suite`, only the tests in the specified suite are going to be run. You can also list existing groups in the YAML file, simply run `protoplaster --list-test-suites test.yaml`.

2.3 Base modules parameters

Each base module requires parameters for test initialization. These parameters describe the tests and are passed to the test class as its attributes.

2.3.1 I2C

Required parameters:

- `bus` - i2c bus to be checked
- `name` - name of device to be detected
- `address` - address of the device to be detected on the indicated bus

2.3.2 GPIO

Required parameters:

- `number` - GPIO pin number
- `value` - value written to that pin

Optional parameters:

- `gpio_name` - name of the sysfs GPIO interface after exporting

2.3.3 Cameras

Required parameters:

- `device` - path to the camera device (eg. /dev/video0)
- `camera_name` - expected camera name
- `driver_name` - expected driver name

Optional parameters:

- `save_file` - a path which the tested frame is saved to (the frame is saved only if this parameter is present)

2.3.4 FPGA

Required parameters:

- `sysfs_interface` - path to a sysfs interface for flashing the bitstream to the FPGA
- `bitstream_path` - path to a test bitstream that is going to be flashed

2.4 Writing additional modules

Apart from base modules available in Protoplaster, you can provide your own extended modules. The module should contain a `test.py` file in the root path. This file should contain a test class that is decorated with `ModuleName("")` from the `protoplaster.conf.module` package. This decorator tells Protoplaster what the name of the module is. With this information, Protoplaster can correctly initialize the test parameters. The test class should contain a `name()` method. Its return value is used for the `device_name` field in CSV output.

The description of the external module should be added to the YAML file as for other tests. By default, external modules are expected in the `/etc/protoplaster` directory. If you want to store them in a different path, use the `--custom-tests` argument to set your own path. Individual tests run by Protoplaster should be present in the main class in the `test.py` file. The class's name should start with `Test`, and every test's name in this class should also start with `test`. An example of an extended module test:

```
from protoplaster.conf.module import ModuleName


@ModuleName("additional_camera")
class TestAdditionalCamera:
    """
    {% macro TestAdditionalCamera(prefix) -%}
    Additional camera tests
    -----------------------
    {% do prefix.append('') %}
    This module provides tests dedicated to camera sensors on specific video node:
    {%- endmacro %}
    """

    def test_exists(self):
        """
```

(continues on next page)

(continued from previous page)

```
        {% macro test_exists(device) -%}
          check if the path exists
        {%- endmacro %}
        """
        assert self.path == "/dev/video0"
```

And a YAML definition:

```
---
base:
  additional_camera:
    - path: "/dev/video0"
    - path: "/dev/video1"
```

2.5 Protoplaster test report

Protoplaster provides `protoplaster-test-report`, a tool to convert test CSV output into a HTML or Markdown table.

```
usage: protoplaster-test-report [-h] [-i INPUT_FILE] -t {md,html} [-o OUTPUT_FILE]

options:
  -h, --help            show this help message and exit
  -i INPUT_FILE, --input-file INPUT_FILE
                        Path to the csv file
  -t {md,html}, --type {md,html}
                        Output type
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
                        Path to the output file
```

2.6 System report

Protoplaster provides `protoplaster-system-report`, a tool for obtaining information about system state and configuration. It executes a list of commands and saves their outputs. The outputs are stored in a single zip archive along with an HTML summary.

2.6.1 Usage

```
usage: protoplaster-system-report [-h] [-o OUTPUT_FILE] [-c CONFIG] [--sudo]

options:
  -h, --help            show this help message and exit
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
                        Path to the output file
  -c CONFIG, --config CONFIG
                        Path to the YAML config file
  --sudo                Run as sudo
```

The YAML config contains a list of actions to perform. A single action is described as follows:

```
report_item_name:
  run: script
  summary:
    - title: summary_title
      run: summary_script
  output: script_output_file
  superuser: required | preferred
  on-fail: ...
```

- `run` - command to be run
- `summary` – a list of summary generators, each one with fields:
  - `title` – summary title
  - `run` – command that generates the summary. This command gets the output of the original command as stdin. This field is optional; if not specified, the output is placed in the report as-is.
- `output` - output file for the output of `run`.
- `superuser` – optional, should be specified if the command requires elevated privileges to run. Possible values:
  - `required` – `protoplaster-system-report` will terminate if the privilege requirement is not met
  - `preferred` – if the privilege requirement is not met, a warning will be issued and this particular item won't be included in the report
- `on-fail` – optional description of an item to run in case of failure. It can be used to run an alternative command when the original one fails or is not available.

Example config file:

```
uname:
  run: uname -a
  summary:
    - title: os info
      run: cat
  output: uname.out
dmesg:
  run: dmesg
  summary:
    - title: usb
      run: grep usb
    - title: v4l
      run: grep v4l
  output: dmesg.out
  superuser: required
ip:
  run: ip a
  summary:
    - title: Network interfaces state
      run: python3 $PROTOPLASTER_SCRIPTS/generate_ip_table.py "$(cat)"
```

(continues on next page)

(continued from previous page)

```
  output: ip.out
  on-fail:
    run: ifconfig -a
    summary:
      - title: Network interfaces state
        run: python3 $PROTOPLASTER_SCRIPTS/generate_ifconfig_table.py "$(cat)"
    output: ifconfig.out
```

2.6.2 Running as root

By default, sudo doesn't preserve `PATH`. To run `protoplaster-system-report` installed by a non-root user as root, invoke `protoplaster-system-report --sudo`

2.7 Protoplaster manager

Protoplaster provides `protoplaster-mgmt`, a tool to remotely control Protoplaster via the API. For more detailed information, see the help messages associated with each subcommand.

```
usage: protoplaster-mgmt [-h] [--url URL] [--config CONFIG] [--config-dir CONFIG_
↪DIR] [--report-dir REPORT_DIR] [--artifact-dir ARTIFACT_DIR] {configs,runs} ...

Tool for managing Protoplaster via remote API

options:
  -h, --help            show this help message and exit
  --url URL             URL to a device running Protoplaster server (default:␣
↪http://127.0.0.1:5000/)
  --config CONFIG       Config file with values for url, config-dir, report-dir,␣
↪artifact-dir
  --config-dir CONFIG_DIR
                        Directory to save fetched config (default: ./)
  --report-dir REPORT_DIR
                        Directory to save a test report (default: ./)
  --artifact-dir ARTIFACT_DIR
                        Directory to save a test artifact (default: ./)

available commands:
  {configs,runs}
    configs             Configs management
    runs                Test runs management
```

CHAPTER

THREE

PROTOPLASTER SERVER API REFERENCE

3.1 Error Handling

Should an error occur during the handling of an API request, either because of incorrect request data or other endpoint-specific scenarios, the server will return an error structure containing a user-friendly description of the error. An example error response is shown below:

```
{
   "error": "test start failed"
}
```

3.2 Configs API

GET /api/v1/configs

Fetch a list of configs

Status Codes

- 200 OK – no error

Response JSON Array of Objects

- **created** (string) – UTC datetime of config upload (RFC822)
- **name** (string) – config name

Example Request

```
GET /api/v1/configs HTTP/1.1
Accept: application/json, text/javascript
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "name": "sample_config.yaml",
    "created": "Mon, 25 Aug 2025 16:58:35 +0200",
  }
]
```

POST /api/v1/configs

Upload a test config

Form Parameters

- **file** – yaml file with the test config

Status Codes

- 200 OK – no error, config was uploaded
- 400 Bad Request – file was not provided

Example Request

```
POST /api/v1/configs HTTP/1.1
Accept: */*
Content-Length: 4194738
Content-Type: multipart/form-data; boundary=------------------------
↪0f8f9642db3a513e

--------------------------0f8f9642db3a513e
Content-Disposition: form-data; name="file"; filename="config.yaml"
Content-Type: application/octet-stream

<file contents>
--------------------------0f8f9642db3a513e--
```

Example Response

```
HTTP/1.1 200 OK
```

DELETE /api/v1/configs/(string: *config_name*)

Remove a test config

Parameters

- **name** – filename of the test config

Status Codes

- 200 OK – no error, config was removed
- 404 Not Found – file was not found

Example Request

```
DELETE /api/v1/configs/sample_config.yaml HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
```

GET /api/v1/configs/(string: *config_name*)

Fetch information about a config

Status Codes

- 200 OK – no error

- 404 Not Found – config does not exist

Response JSON Object

- **created** (string) – UTC datetime of config upload (RFC822)
- **config_name** (string) – config name

Example Request

```
GET /api/v1/configs/sample_config.yaml HTTP/1.1
Accept: application/json, text/javascript
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "name": "sample_config.yaml",
  "created": "Mon, 25 Aug 2025 16:58:35 +0200",
}
```

GET /api/v1/configs/(string: *config_name*)/file

Fetch a config file

Status Codes

- 200 OK – no error
- 404 Not Found – config does not exist

>file text/yaml

YAML config file

Example Request

```
GET /api/v1/configs/sample_config.yaml/file HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: text/yaml
Content-Disposition: attachment; filename="sample_config.yaml"

base:
  network:
    - interface: enp14s0
```

3.3 Test Runs API

GET /api/v1/test-runs

Fetch a list of test runs

Status Codes

- 200 OK – no error

Response JSON Array of Objects

- **id** (string) – run id
- **config_name** (string) – name of config for this test run
- **test_suite_name** (string) – name of the test suite for this test run
- **created_at** (string) – UTC datetime of test run creation (RFC822)
- **started_at** (string) – UTC datetime of test run start (RFC822)
- **finished_at** (string) – UTC completion time (RFC822)
- **status** (string) – test run status, one of: * pending - accepted but not started * running - currently executing * finished - completed successfully * failed - error during execution * aborted - stopped by user or system
- **metadata** (dict[str, str]) – optional test run metadata (key/value pairs)

Example Request

```
GET /api/v1/test-runs HTTP/1.1
Accept: application/json, text/javascript
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "id": "25d9f4a2-2556-4647-b3cc-762348dc51ce",
    "config_name": "config1.yaml"
    "test_suite_name": "simple-test"
    "status": "finished",
    "created_at": "Mon, 25 Aug 2025 15:56:35 +0200",
    "started_at": "Mon, 25 Aug 2025 15:56:36 +0200",
    "finished_at": "Mon, 25 Aug 2025 15:56:44 +0200",
    "metadata": {
      "bsp-sha256":
↪"a5603553e0eaad133719dc19b57c96e811a72af5329e120310f96b4fdc891732"
    }
  },
  {
    "run_id": "976c3d37-0b9a-4c81-ad0d-ebb96c9eee94",
    "config_name": "config2.yaml"
    "test_suite_name": "complex-test"
    "status": "running",
    "created_at": "Mon, 25 Aug 2025 16:58:35 +0200",
    "started_at": "Mon, 25 Aug 2025 16:58:36 +0200",
    "finished_at": "",
```

(continues on next page)

(continued from previous page)

```
    "metadata": {}
  }
]
```

POST /api/v1/test-runs

Trigger a test run

Status Codes

- 200 OK – no error, test run was triggered
- 404 Not Found – config file was not found

Request JSON Object

- **config_name** (string) – name of config for this test run
- **test_suite_name** (string) – name of the test suite for this test run

Example Request

```
POST /api/v1/test-runs/ HTTP/1.1
Content-Type: application/json
Accept: application/json, text/javascript

{
  "config_name": "config1.yaml",
  "test_suite_name": "simple-test"
}
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": "25d9f4a2-2556-4647-b3cc-762348dc51ce",
  "config_name": "config1.yaml"
  "test_suite_name": "simple-test"
  "status": "pending",
  "created_at": "Mon, 25 Aug 2025 15:56:35 +0200",
  "started_at": "",
  "finished_at": "",
  "metadata": {}
}
```

DELETE /api/v1/test-runs/(string: *identifier*)

Cancel a pending test run

Parameters

- **identifier** – test run identifier

Status Codes

- 200 OK – no error

- 400 Bad Request – test run not pending
- 404 Not Found – test run does not exist

Example Request

```
DELETE /api/v1/test-runs/25d9f4a2-2556-4647-b3cc-762348dc51ce HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "run_id": "25d9f4a2-2556-4647-b3cc-762348dc51ce",
  "config_name": "config1.yaml"
  "test_suite_name": "simple-test"
  "status": "aborted",
  "created_at": "Mon, 25 Aug 2025 15:56:35 +0200",
  "finished_at": "Mon, 25 Aug 2025 15:56:44 +0200",
  "metadata": {
    "bsp-sha256":
↪"a5603553e0eaad133719dc19b57c96e811a72af5329e120310f96b4fdc891732"
  }
}
```

GET /api/v1/test-runs/(string: *identifier*)

Fetch information about a test run

Parameters

- **identifier** – test run identifier

Status Codes

- 200 OK – no error
- 404 Not Found – test run does not exist

Response JSON Object

- **id** (string) – test run identifier
- **config_name** (string) – name of config for this test run
- **test_suite_name** (string) – name of the test suite for this test run
- **created_at** (string) – UTC creation time (RFC822)
- **started_at** (string) – UTC creation time (RFC822)
- **finished_at** (string) – UTC completion time (RFC822)
- **status** (string) – test run status, one of: * pending - accepted but not started * running - currently executing * finished - completed successfully * failed - error during execution * aborted - stopped by user or system
- **metadata** (dict[str, str]) – optional test run metadata (key/value pairs)

Example Request

```
GET /api/v1/test-runs/25d9f4a2-2556-4647-b3cc-762348dc51ce HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "run_id": "25d9f4a2-2556-4647-b3cc-762348dc51ce",
  "config_name": "config1.yaml"
  "test_suite_name": "simple-test"
  "status": "finished",
  "created_at": "Mon, 25 Aug 2025 15:56:35 +0200",
  "started_at": "Mon, 25 Aug 2025 15:56:36 +0200",
  "finished_at": "Mon, 25 Aug 2025 15:56:44 +0200",
  "metadata": {
    "bsp-sha256":
↪"a5603553e0eaad133719dc19b57c96e811a72af5329e120310f96b4fdc891732"
  }
}
```

GET /api/v1/test-runs/(string: *identifier*)**/artifacts**

Fetch a list of test run artifacts.

Parameters

- **identifier** – test run identifier

Status Codes

- 200 OK – no error, file returned
- 404 Not Found – test run not completed or does not exist

>file

artifact file with content type inferred automatically

Example request

```
GET /api/v1/runs/25d9f4a2-2556-4647-b3cc-762348dc51ce/artifacts HTTP/1.1
```

Example response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "name": "frame.raw",
    "created": "Mon, 25 Aug 2025 16:58:35 +0200",
  }
]
```

GET /api/v1/test-runs/(string: *identifier*)**/artifacts/**
path: *artifact_name*

Fetch test run artifact.

Parameters

- **identifier** – test run identifier
- **artifact_name** – artifact filename

Status Codes

- 200 OK – no error, file returned
- 404 Not Found – test run not completed or does not exist, or artifact does not exist

>file
artifact file with content type inferred automatically

Example request

```
GET /api/v1/runs/25d9f4a2-2556-4647-b3cc-762348dc51ce/artifacts/frame.raw␣
↪HTTP/1.1
```

Example response

```
HTTP/1.1 200 OK
Content-Type: <depends on artifact>
Content-Disposition: attachment; filename="frame.raw"
```

GET /api/v1/test-runs/(string: *identifier*)**/report**

Fetch test run report

Parameters

- **identifier** – test run identifier

Status Codes

- 200 OK – no error
- 404 Not Found – test run not completed or does not exist, or report file does not exist

>file text/csv
CSV file containing the full test report

Example request

```
GET /api/v1/test-runs/25d9f4a2-2556-4647-b3cc-762348dc51ce/report HTTP/1.1
```

Example response

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="25d9f4a2-2556-4647-b3cc-
↪762348dc51ce.csv"
```

(continues on next page)

(continued from previous page)

```
device name,test name,module,duration,message,status
enp14s0,exist,test.py::TestNetwork::test_exist,0.0007359918672591448,,passed
```

CHAPTER

FOUR

WEB UI

When Protoplaster is running in server mode, it serves a Web UI which can be used for remote configuration and tests triggering. It also supports controlling other devices running Protoplaster in server mode from a single Web UI.

4.1 Accessing Web UI

Run Protoplaster in server mode, optionally specifying a port:

```
protoplaster -d tests/ -r reports/ -a artifacts/ --server --port 5000
```

Access the Web UI from a web browser using the device's IP and the specified port:

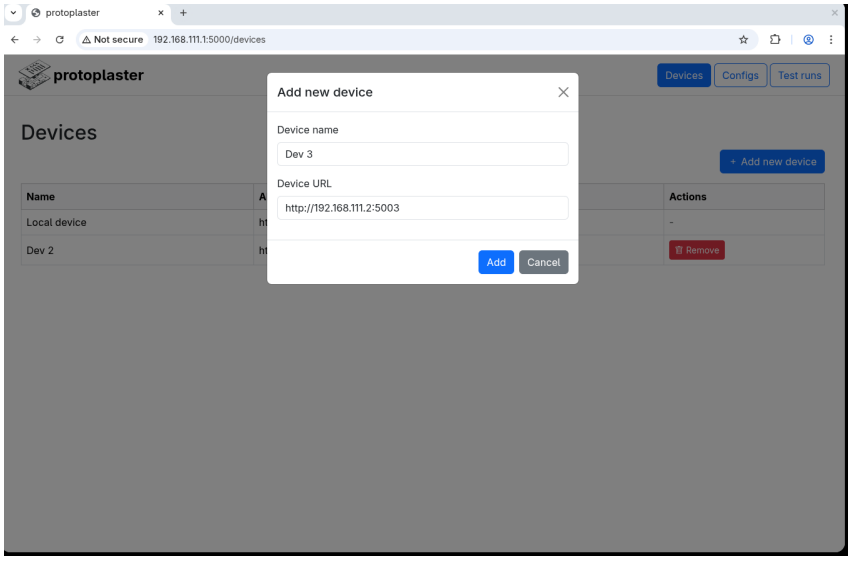

Figure 4.1: Protoplaster Web UI.

4.2 Devices

In the Devices tab, external devices can be added and removed. The local device (the one serving the Web UI) is always present and cannot be removed. To add an external device, first start Protoplaster on it in server mode, then add it using the "Add new device" button, specifying its IP and port on which Protoplaster is running:

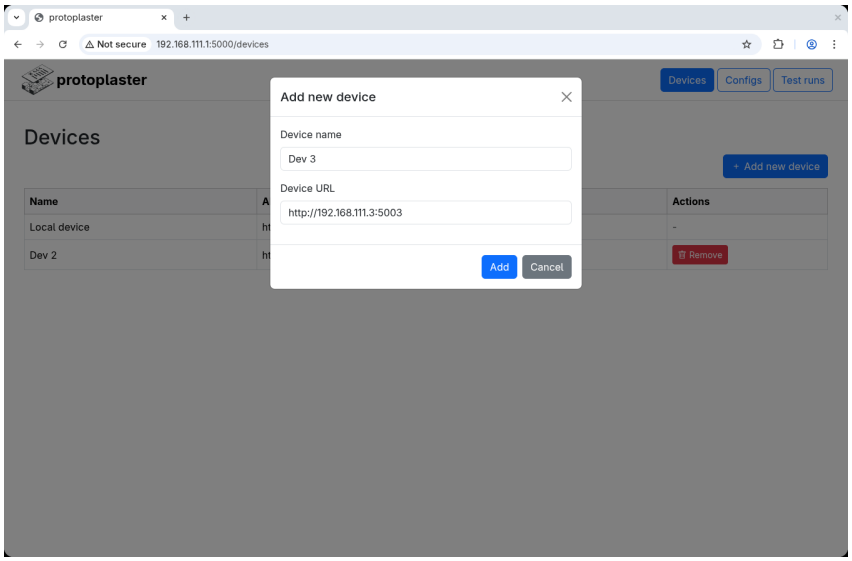

Figure 4.2: Web UI - adding new device.

4.3 Configs

In the Configs tab, all the available configs can be listed for all connected devices. Select the device for which configs are listed using the drop-down on the left. To add a new config, use the "Add new config" button. Configs can be uploaded to multiple devices by selecting more than one in the Devices box:

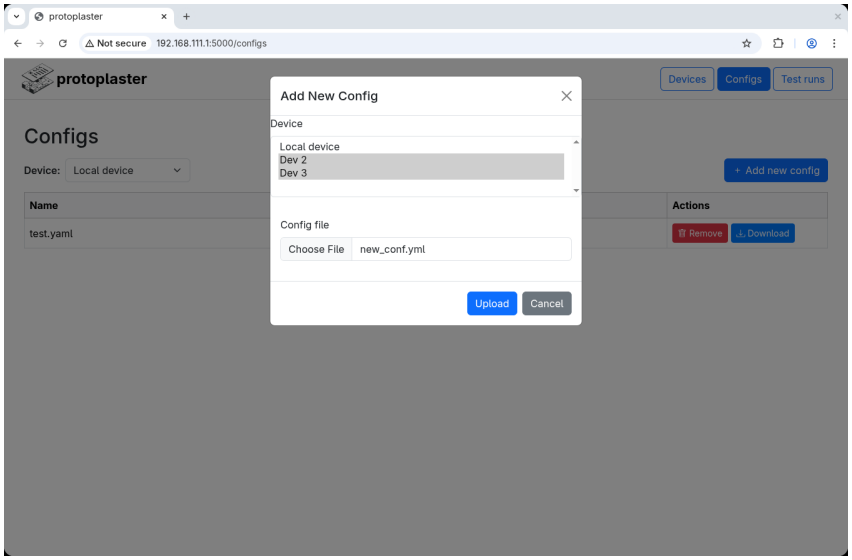

Figure 4.3: Web UI - adding new config.

4.4 Test runs

In the Test runs tab, running and finished test runs can be listed for available devices. Like in the Configs tab, the desired device for which test runs are listed can be selected using the drop-down menu. To trigger a test run, use the "Trigger test run" button. Tests can be triggered on multiple devices, by selecting multiple devices in the Devices box. When multiple devices are selected, the config field will only show configs which are present on all selected devices.

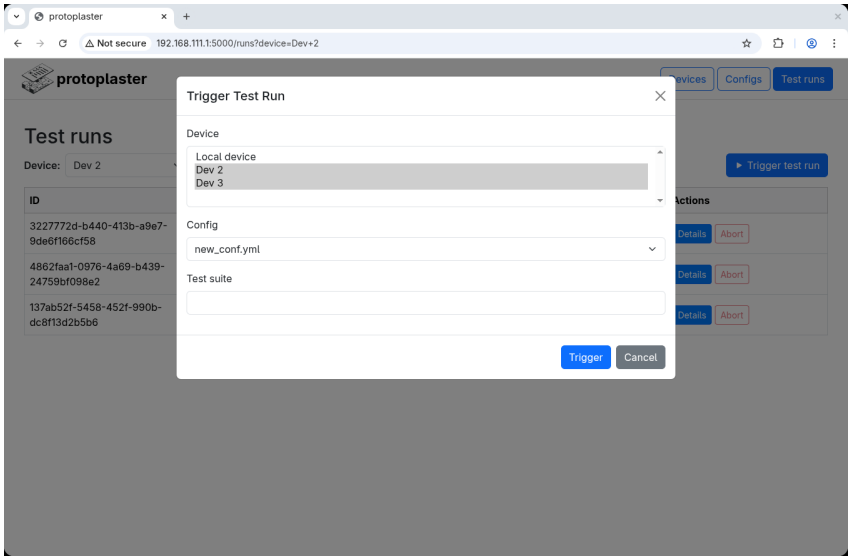

Figure 4.4: Web UI - adding new config.

CHAPTER

FIVE

PROTOPLASTER TESTS

Note

This page has been autogenerated from a Protoplaster tests definition file.

To perform hardware/BSP tests and open-source Protoplaster framework has been used.

Running Protoplaster runs the tests described in the following chapters:

5.1 I2C devices tests

This module provides tests dedicated to i2c devices on specific buses:

- /dev/i2c-0:
  - detection test for *Sensor name* on address: `0x3c`
- /dev/i2c-0:
  - detection test for *I2C-bus multiplexer* on address: `0x70`

5.2 GPIOs tests

This module provides tests dedicated to GPIO on specific pin number

- /sys/class/gpio/gpio20:
  - write `1` and read back to confirm

5.3 Camera sensor tests

This module provides tests dedicated to V4L devices on specific video node:

- vivid:
  - try to capture frame
  - check if the camera sensor name is `vivid`
  - check if the camera sensor driver name is `vivid`
- vivid:
  - try to capture frameand store it to `frame.raw` file

- check if the camera sensor name is `vivid`
- check if the camera sensor driver name is `vivid`

CHAPTER

SIX

PROTOPLASTER TESTS REPORT

device name	test name	module	duration	message	status	artifacts
/dev/i2c-0	address	test.py:	15m 480u	AssertionError: No device found at address: 60assert False + where False = <bound method I2C.check_address of <protoplaster.tests.i2c.i2c.I2C object at 0x3fdf4fd510>>(60, True) + where <bound method I2C.check_address of <protoplaster.tests.i2c.i2c.I2C object at 0x3fdf4fd510>> = <protoplaster.tests.i2c.i2c.I2C object at 0x3fdf4fd510>.check_address	failed	['uname']
/dev/i2c-0	fram	test.py:	232m 601u		passed	['uname']
/dev/i2c-0	device_	test.py:	208m 881u		passed	['uname']
/dev/i2c-0	drive	test.py:	205m 89us		passed	['uname']
/sys/bus/i2c/devices/i2c-0/name	read_	test.py:	12m 561u		passed	['uname']

CHAPTER

SEVEN

PROTOPLASTER SYSTEM REPORT

Protoplaster provides `protoplaster-system-report`, a tool to obtain information about system state and configuration. It executes a list of commands and saves their outputs. The outputs are stored in a single zip archive together with an HTML summary. An example summary can be found here.

The following config was used to generate the example:

```
uname:
  run: uname -a
  summary:
    - title: os info
      run: cat
  output: uname.out
dmesg:
  run: dmesg
  summary:
    - title: usb
      run: grep usb
    - title: v4l
      run: grep v4l
  output: dmesg.out
  superuser: required
ip:
  run: ip a
  summary:
    - title: Network interfaces state
      run: python3 $PROTOPLASTER_SCRIPTS/generate_ip_table.py "$(cat)"
  output: ip.out
  on-fail:
    run: ifconfig -a
    summary:
      - title: Network interfaces state
        run: python3 $PROTOPLASTER_SCRIPTS/generate_ifconfig_table.py "$(cat)"
    output: ifconfig.out
```

/api