



Using MTDA for testing (Multi-Tenant Device Access)

Open Source @ Siemens 2022

Cedric Hombourger

*Siemens Digital Industries Software
Embedded Platform Solutions group*

Problem Statements



Platform team needing to support many different reference/customer boards and switch between them frequently/quickly



More Work From Home (WFH) than before: need to limit hardware we take into home offices (less sharing of hardware between team members)



Needing to remotely install our OS builds with all security measures in place (e.g., SecureBoot): NFS boot no longer an option

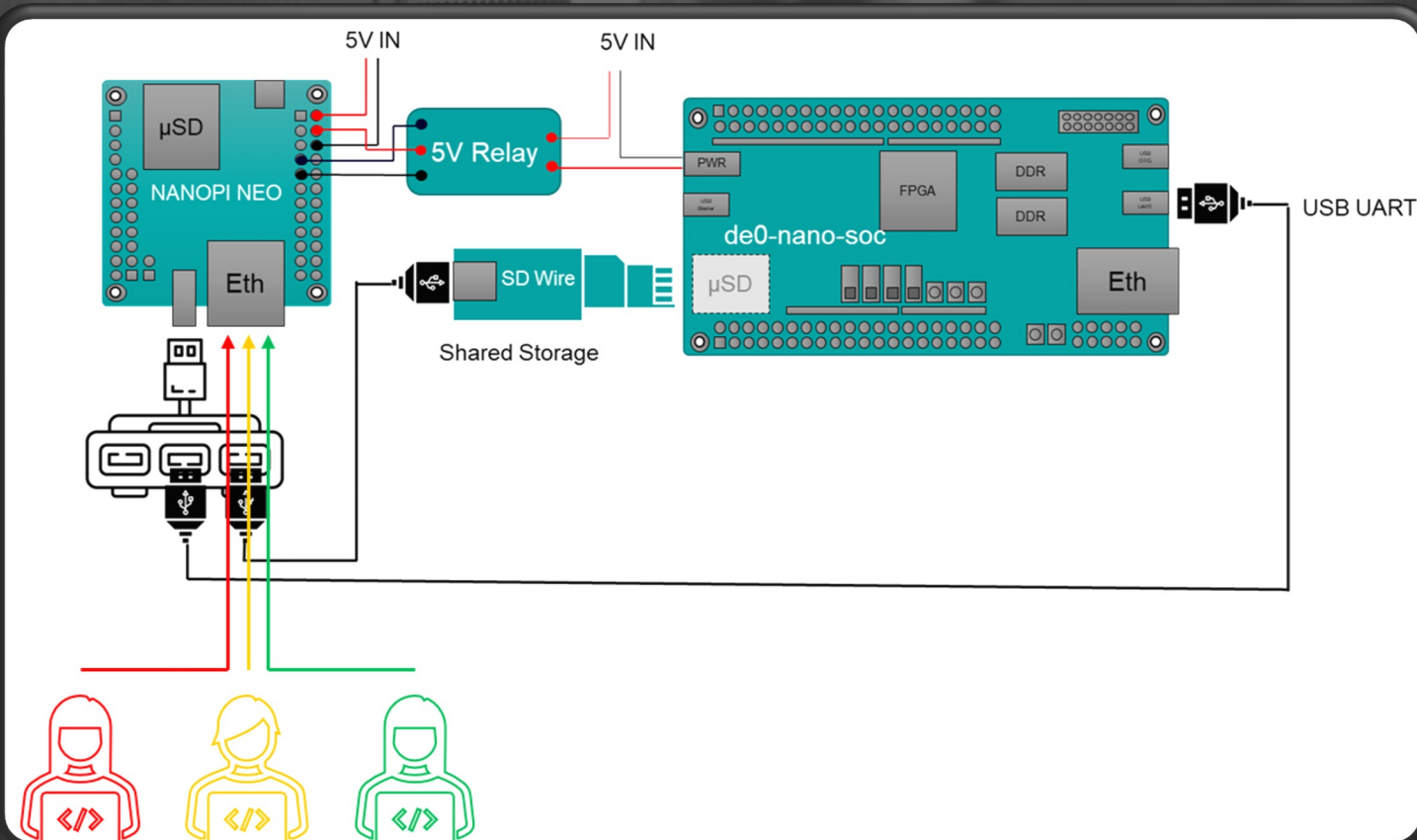


Needing development and QA teams to use similar deployment workflows to reduce ping-ping games (developer: “works for me” VS tester: “still seeing the issue over here”)

Introducing MTDA

- MTDA is a shorthand for Multi-Tenant Device Access: a Python tool for multiple folks to access the same hardware device
- Initially created in 2017 for my own usage: limited office space, various hardware boards to work with and tired to commute back-and-forth between my desk and my hardware lab!
- Started small with a small Python application implementing 4 functions:
 - Power control using a controllable Power outlet from Aviosys
 - Buffering of serial console output into a ring buffer
 - Swapping of SD Card between host and target using Samsung's SD_MUX (a small USB hardware)
 - Minimalistic terminal UI supporting the above features
- Advertised the project internally ("hey I have created this, what do you think?) and got interest from our QA team
- Project Open Sourced in late 2021: <https://github.com/siemens/mtda>

Example deployment



- Terasic de0-nano-soc as Device Under Test
- Use of the NanoPI NEO as a proxy / gateway to the Device Under Test
- De0-nano-soc boots from SD-Card, using SD Wire to program it from the NanoPI NEO
- We may interact with the de0-nano-soc using its debug UART
- Multiple users may access the de0-nano-soc through the NanoPI NEO. They all see the same console at the same time and control it. It is possible for one of them to grab a lock, other sessions will then become read-only

Example workflow

```
$ export MTDA_REMOTE=mtda-for-pi4.mydomain.net
```

```
$ mtda-cli target off # remotely power off the Pi4
```

```
$ mtda-cli storage host # swap SD card to Assist Board
```

```
$ mtda-cli storage write my-os-image.wic # write my OS build (Yocto/ISAR) to the SD card
```

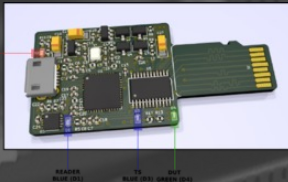
```
$ mtda-cli storage target # attach SD card back to Target Board
```

```
$ mtda-cli target on # power the Pi4, it will boot from the SD card we have just programmed
```

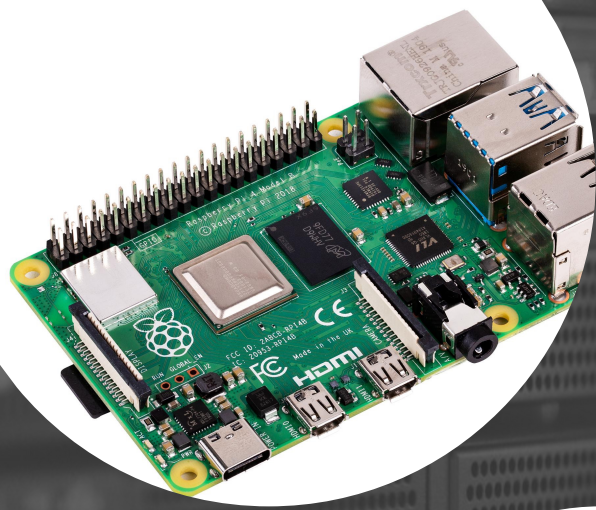
```
$ mtda-cli console interactive # see console output
```

How does it work?

- The Assist Board is running mtda-cli in “daemon” mode and has 2 ZeroRPC sockets:
 - A control socket (target_off, target_on, storage_to_host, storage_to_target, etc.)
 - A console socket to publish console output to clients (using a ZeroRPC Publish/Subscribe socket)
- Power commands are received on the control socket and are simply forwarded to the selected “power driver”
 - GPIO pull up/down when using relays
 - pduclient --command on/off for setups controlled by LAVA
- Storage commands are also received on the control socket and forwarded to the selected “storage driver”:
 - Write to local file/partition for Target boards that may be boot from USB (the assist board can present itself as a Mass Storage device using Linux USB gadget stack => Assist board needs to have a USB OTG/Function port)
 - Write to SD Card using SDWire hardware



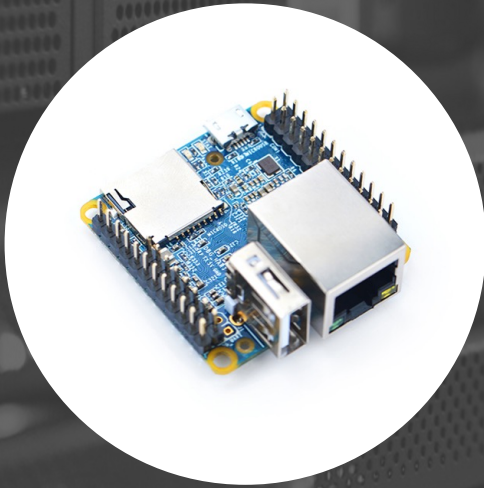
- Console output is published by the console logger thread which reads from the selected UART and writes to the Publish/Subscribe socket (using the “CON” topic/channel)
- Console input is just another “control command” where we simply write whatever the user types to the selected UART



Assist Boards

Since most laptops / PCs do not have a USB Function/OTG port, we use a small factor board as “gateway”

The following boards are supported:



- NanoPI NEO LTS (cheap!)
- NanoPI R1
- Raspberry Pi4 (kudos to Jan Kiszka)
- BeagleBone Black



The MTDA image running on those devices may be built using kas (<https://github.com/siemens/kas>) and uses ISAR (<https://github.com/ilbers/isar>) to produce an image based on Debian 11

Not seeing your favorite SBC? Feel free to contribute your own Board Support package to the project ☺
(we also accept hardware donations, e.g., the NanoPI R1 was donated by Gunther Birk)

Target Boards

We have used MTDA with the following target boards:

- DE0-NANO - Terasic
- i.MX6 Sabre Lite - Boundary Devices
- i.MX8 MQ-EVK – NXP
- IOT2050 - Siemens
- Pi4 – Raspberry
- SIMATIC IPC127e – Siemens
- SIMATIC IPC227e – Siemens
- SIMATIC IPC427e - Siemens
- ZCU102 – Xilinx
- ZCU106 – Xilinx
- And more... (e.g., customer hardware we cannot name here ☺)

Control Drivers

The following driver categories:

- power - control power delivery to the Device Under Test
- storage - to swap storage between the host and target
- console - to interact with the target console over UART
- usb - to switch USB ports ON/OFF
- keyboard - to emulate key presses
- video - to stream video output from the Device Under Test to clients

Configuration file

Here's a sample config(ini) file for the Assist Board:

```
# make the assist board expose a virtual UART over USB
[console]
variant=usbf
```

```
# control a 5V relay using GPIO #203
[power]
variant=gpio
pins=203
```

```
# emulate a USB keyboard
[keyboard]
variant=hid
device=/dev/hidg0
```

```
# emulate a USB stick (to boot from)
[storage]
variant=usbf
file=/var/lib/mta/storage.img
```


Power Drivers

Control power delivery

Power drivers need to implement the following methods:

- `configure()` # get settings from our configuration files
- `probe()` # determine if the selected “power” control device is there (when possible)
- `on()` # deliver power to the Device Under Test
- `off()` # cut power from the Device Under Test
- `status()` # determine if power is being delivered to the Device Under Test
- `wait()` # wait for a power change event

The following drivers are supported:

- `aviosys_8800` – a regular power outlet that may be controlled over USB
- `docker` – use when the “device” under test is a docker container
- `gpio` – control 5V relays over GPIO lines
- `pduclient` – use LAVA power control interface
- `qemu` – use when the “device” under test is a QEMU/KVM instance
- `shellcmd` – run a user-specified command to toggle power
- `usbrelay` – control a relay supported by the `usbrelay` command



(Shared) Storage Drivers

Swap storage between host and target

Storage drivers need to implement the following methods:

- `configure()` # get settings from our configuration files
- `probe()` # determine if the selected “storage” sharing device is there (when possible)
- `to_host()` # transfer shared storage to host (Assist Board)
- `to_target()` # transfer shared storage to target (Device Under Test)
- `status()` # determine if the shared storage is with the host or target
- `open()`, `write()`, `close()` # write data to the shared storage

The following drivers are supported:

- `docker` – use when the “device” under test is a docker container
- `qemu` – use when the “device” under test is a QEMU/KVM instance
- `samsung` – use Samsung SDMux or SDWire to swap SD-Card between host and target
- `usbf` – use Linux USB gadget stack on the Assist Board to emulate a USB Mass Storage device for targets that may boot from USB

Console Drivers

UART console for headless interaction with targets

Console drivers need to implement the following methods:

- `configure()` # get settings from our configuration files
- `probe()` # determine if the selected “console” device is there (when possible)
- `open()`, `close()` # open/close the selected UART (e.g. `/dev/ttyUSB0`)
- `pending()` # return number of bytes that may be consumed from the opened UART
- `read()`, `write()` # read/write bytes from/to the opened UART

The following drivers are supported:

- `docker` – use when the “device” under test is a docker container
- `qemu` – use when the “device” under test is a QEMU/KVM instance
- `serial` – use a regular serial device (`/dev/ttyS0`, `/dev/ttyUSB0`, etc.)
- `telnet` – connect to a remote UART port using the telnet protocol
- `usbf` – use Linux USB gadget stack on the Assist Board to emulate a USB Serial device for targets without UART ports

Switch USB devices ON/OFF

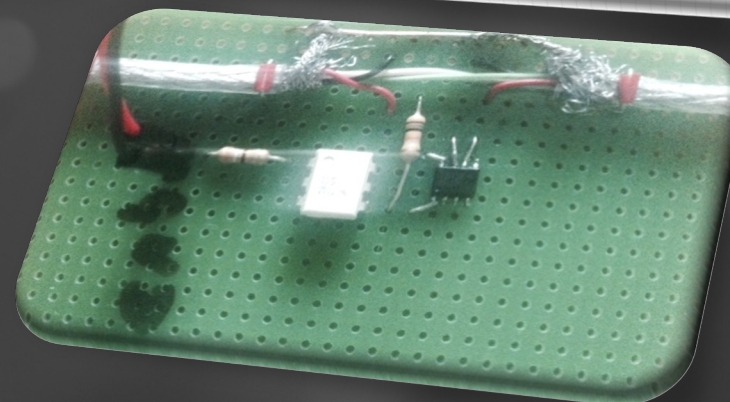
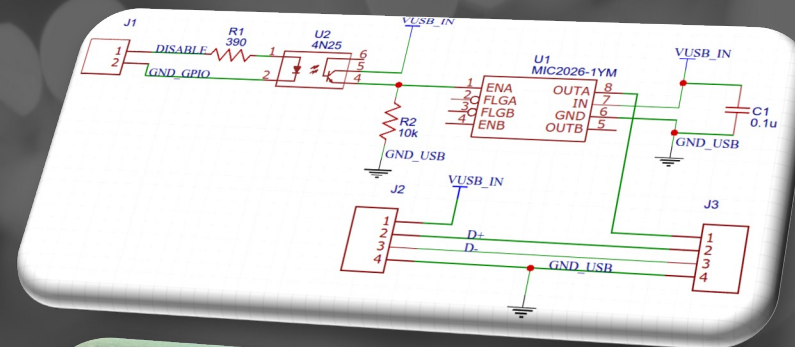
With USB2, we may just toggle the VBUS line ON/OFF for a USB device to be seen by the Device Under Test. This can be done with a relatively simple piece of hardware. We use this to programmatically attach USB devices we routinely have to test (e.g., 4G modems)

USB drivers need to implement the following methods:

- `configure()` # get settings from our configuration files
- `probe()` # determine if the selected USB switching device is there (when possible)
- `on()`, `off()` # toggle VBUS for the selected USB port
- `status()` # determine the VBUS state of the selected USB port

The following drivers are supported:

- **qemu** – use when the “device” under test is a QEMU/KVM instance
- **gpio** – control the VBUS line of a USB port using GPIO lines



Keyboard Drivers

Emulate key presses

For some devices to boot from e.g., USB, it is sometimes necessary to hit a few keys shortly after the device is powered on to boot from USB. This is typically used in “power on” scripts. Keys may also be sent after the device was fully booted.

Keyboard drivers need to implement the following methods:

- `configure()` # get settings from our configuration files
- `probe()` # determine if the selected virtual “keyboard” device is there (when possible)
- `esc()`, `f1()..f12()`, `up()`, `left()`, `right()`, `down()`, `enter()` # send key

The following drivers are supported:

- `hid` – use the Linux USB gadget stack to emulate a USB keyboard
- `qemu` – use when the “device” under test is a QEMU/KVM instance

Video Drivers

Stream video output from the DUT to MTDA clients

While many test cases may be executed and checked over the serial console, support for graphical applications may be desirable. Many reference boards provide a HDMI or DisplayPort interface. Using a capture device, we may get a video stream. MTDA clients such as test applications could process the video stream using OpenCV (or similar) to programmatically check the output.

Video drivers need to implement the following methods:

- `configure()` # get settings from our configuration files
- `probe()` # determine if the “video” capture device is there (when possible)
- `start()`, `stop()` # start/stop video capture
- `url()` # return the URL clients may use to view the video stream

The following drivers are supported:

- `mjpg_streamer` – use the `mjpg_streamer` tool to stream video using V4L2
- `qemu` – use when the “device” under test is a QEMU/KVM instance



Beyond Interactive Usage

While Developers would mostly use MTDA interactively to remotely access, program and control reference/custom boards, it may also be used with:

- Pytest - create unit tests using Python where a test image is programmed, and commands sent over the UART interface to check if the Operating System behaves as expected
- LAVA - use MTDA to program the Device Under Test and then LAVA for everything else (via the console interface provided by MTDA)
- Your browser - quickly access a reference/custom board using your browser to check a few things
- QEMU/KVM - offline / unable to access the lab? MTDA may be configured to use QEMU/KVM to emulate a PC with power on/off, USB on/off, shared storage, video streaming, TPM, etc. May also be used to develop Pytest units locally

Runtime tests using Pytest

The Pytest framework makes it easy to write tests in Python and exercise your software stack. MTDA provides support classes to interact with your device and verify its functions using a suite of pytest units. MTDA API tests found in the [tests](#) folder may be used as examples.

Here's a sample test:

```
from mtda.pytest import Console
from mtda.pytest import Target

def test_console_wait_for(powered_off):
    # Power on and wait for login prompt
    # with a timeout of 5 minutes
    assert Target.on() is True
    assert Console.wait_for("login:", timeout=5*60) is not None
```

Noticed the use of a test fixture named [powered_off](#)? Test fixtures may be used to setup the test case and may be defined in [conftest.py](#) (see next slide)

Writing test fixtures for Pytest

A sample test fixture is provided below:

```
from mtda.pytest import Target
from mtda.pytest import Test
```

```
@pytest.fixture()
def powered_off():
    Test.setup()
    assert Target.off() is True

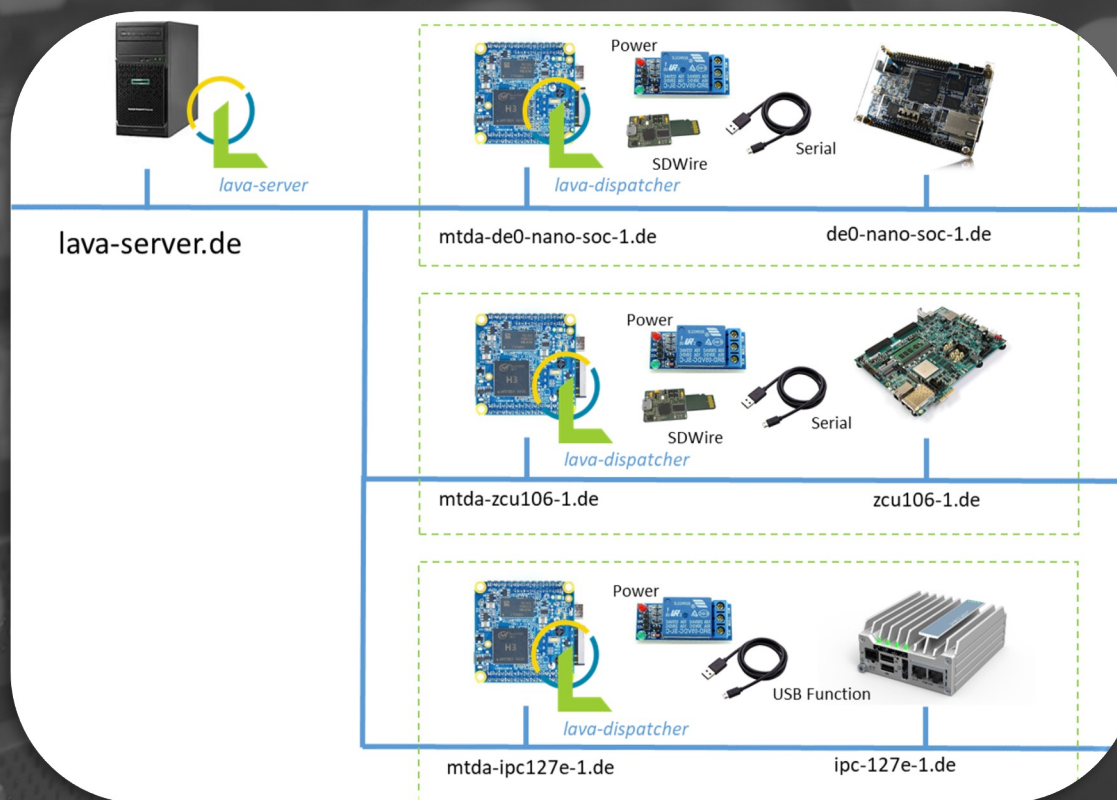
    yield "powered off"

    Test.teardown()
```

Additional test fixtures may be created to get a shell prompt by entering user credentials, to setup networking, to programmatically attach/detach USB devices, etc.

Runtime tests using LAVA

LAVA is a continuous integration platform for deploying and testing operating systems onto physical and virtual hardware. It needs methods to power targets, write system images and interact with the system (usually over a serial console). MTDA can implement these functions. A sample deployment is depicted below:



Device configuration in LAVA using MTDA

A **mtda** device-type may be added to LAVA to define how to power the DUT, how to write the test image and how to connect (interact) with the DUT:

```
{# device_type: mtda #}
{% extends 'base.jinja2' %}

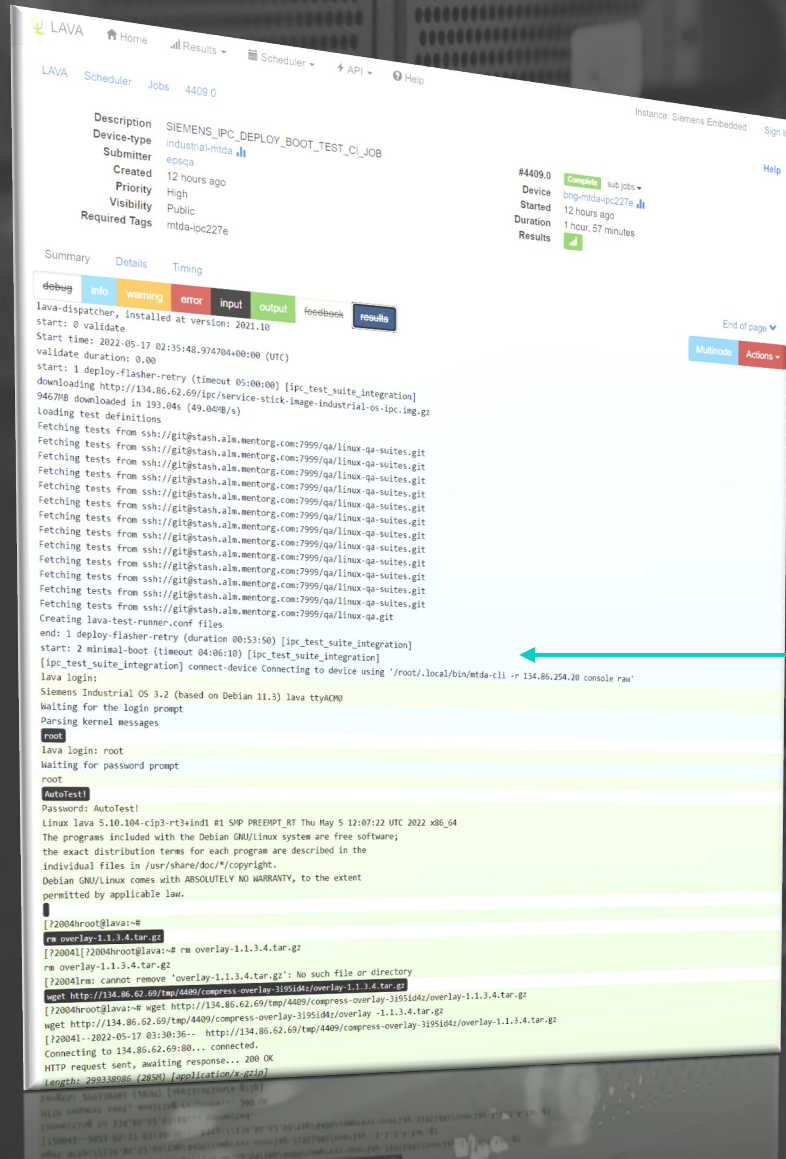
{% set def_mtda_agent = 'localhost' %}
{% set mtda_cli = 'mtda-cli -r ' ~ mtda_agent | default(def_mtda_agent) %}

{% set connection_command = mtda_cli ~ ' console raw' %}
{% set power_off_command = mtda_cli ~ ' target off' %}
{% set power_on_command = mtda_cli ~ ' target on' %}
{% set hard_reset_command = mtda_cli ~ ' target reset' %}

{% set def_mtda_deploy_cmds = [mtda_cli ~ ' target off',
                               mtda_cli ~ ' storage host',
                               mtda_cli ~ ' storage write "{IMAGE}"',
                               mtda_cli ~ ' storage target'] %}

{% block body %}
actions:
  deploy:
    methods:
      flasher:
        commands: {{ mtda_deploy_cmds | default(def_mtda_deploy_cmds) }}
  boot:
    connections:
      serial:
        methods:
          minimal:
{% endblock body %}
```

Sample LAVA Test Run



LAVA runs the following commands:

- mtda-cli target off
- mtda-cli storage host
- mtda-cli storage write service-stick-industrial-os-ipc.img.gz
- mtda-cli storage target
- mtda-cli target on
- mtda-cli console raw

LAVA has STDIN/STDOUT/STDERR from mtda-cli opened to communicate with the Device Under Test (e.g., to login as “root”)

Web-based device access

The image displays a web browser window with a Siemens login interface. The browser's address bar shows the URL `http://192.168.1.36:5000`. The login page features the Siemens logo and the text "SIMATIC WinCC OA". The login form includes fields for "User" and "Password", and a checkbox for "Store Password". The URL `https://localhost:443 (EdgeBox@eb012-ppmd.siemens.net)` is visible above the input fields. The browser's status bar at the bottom indicates "100%" zoom and the time "11:46 AM".

Overlaid on the left side of the browser window is a terminal window showing system logs. The logs include timestamps and error messages, such as "Error: Transport error: Cannot obtain token: Error: getaddrinfo ENOTFOUND eb008-ppmd.siemens.net" and "Error: Connection error code=500, data=undefined". The terminal window also shows system information like "uptime" and "load average".

Below the terminal window, the text "Interactive console over UART" is displayed in a teal color.

Below the browser window, the text "Video stream using HDMI capture" is displayed in a teal color.

On the right side of the browser window, the text "MTDA" is visible vertically.

The Siemens logo is located in the bottom right corner of the image.

Measuring Boot Time

mtda will reset its stopwatch when the Device Under Test is powered on. Each line printed to the console may be prefixed with a timestamp using the interactive console (`mtda-cli console interactive` and then `Ctrl-A T`). It is also possible to auto start/stop printing of timestamps via configuration:

```
[console]
variant=serial
device=/dev/ttyUSB0
time-from = U-Boot
time-until = login:
```

mtda would then start its stopwatch as soon as **U-Boot** is printed to the console (instead of starting it when the device is powered on) and stop it when we get a **login:** prompt

Wish List

Here are few things that could make MTDA better:

- Mouse/touch driver – support sending (virtual) mouse/touch events to the DUT
- Sound capture driver – support use of the Audio IN from the Assist board to capture audio from the DUT
- Wi-Fi driver – support use of the Wi-Fi interface the Assist board to create or connect to a test network
- Better Web UI – provide more controls, check why console is sluggish
- Code reviews – there are areas where we could use a better design or code
- Replace ZeroRPC – <https://github.com/Orpc/zerorpc-python> a bit stalled, switch to MQTT?
- Support Yocto BSPs –being able to quickly support more SBCs as Assist Boards by using Yocto to build images for them
- LAVA docker image – provide a Dockerfile to build an image with LAVA pre-configured

Contributing

We use GitHub for Pull Requests (<https://github.com/siemens/mtda/pulls>), issue tracking (<https://github.com/siemens/mtda/issues>) and for discussions (<https://github.com/siemens/mtda/discussions>)

We welcome any contributions. While maintaining MTDA isn't our primary job, it is a tool we use every day and would be very pleased to make it useful for others and used more broadly.

Code contributions will be reviewed online using GitHub. Maintainers will trigger some (basic) automated tests for your changes (test logs/results will show up in the Pull Request). Code changes shall keep pycodestyle happy. Make sure you have tested your feature/fix


Online documentation (<https://mtda.readthedocs.io/en/latest/>) gets updated with every release (via GitHub actions)

Prebuilt Debian packages made available via Gemfury (<https://apt.fury.io/mtda/>) and PPA for Ubuntu (<https://launchpad.net/~chombourger/+archive/ubuntu/mtda-focal>)

Happy Coding, Happy Testing! Looking forward to your contributions!

Thank you!

Special thanks to:

- Siemens for allowing us to open-source this project
- Tizen project for sharing their SDWire design (<https://wiki.tizen.org/SDWire>)
- Early contributors:
 - badrikeshp-mg 
 - bhargavdas 
 - gubi34 
 - hkoturap 
 - jan-kiszka 
 - jjmcdn 
 - ptsarath 
 - ricbha 
 - vj-kumar 