Testing Automation in an Embedded Linux Project

Topi Kuutela

December 22, 2015

Contents

1 Start here

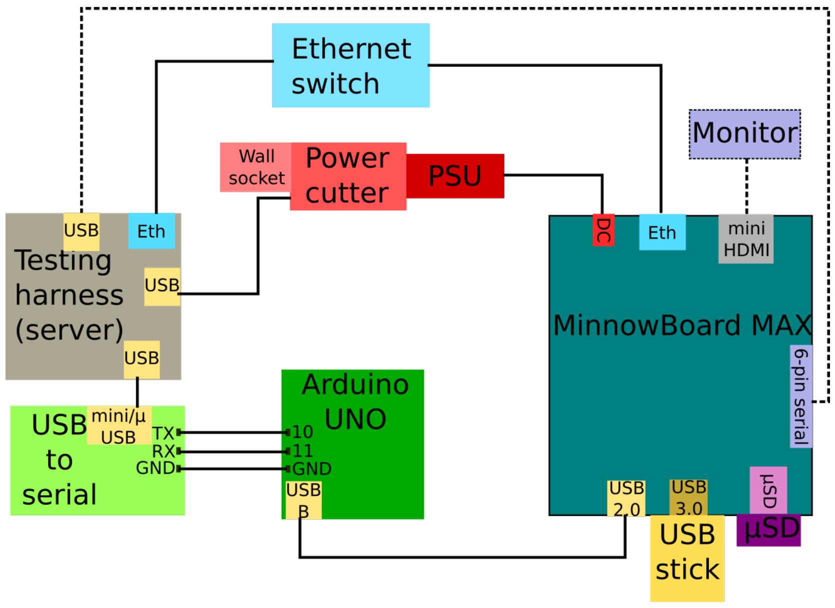


Figure 1: MinnowBoard MAX wiring diagram

1.1 Simple setup

The absolutely mandatory steps to take for a simplest working system with just MinnowBoard MAX platform are as follows:

1. Copy the content of Testing automation USB-stick from ta-usb-stick branch in the AFT git repository to a suitable USB-stick.

   Note: At the time of writing, the location is temporary and doesn't include pre-made Debian images due to their size.

2. Setup a Testing harness as instructed in section 2

3. Pick a Cleware powercutter and using the instructions on section 5.1, install the associated software.

4. Setup AFT as instructed in section 3

5. Setup PEM as instructed in section 4

6. Create an Arduino keyboard emulator using the instructions in 4.1

7. Using `dd` create a bootable Debian USB-stick for MinnowBoard MAX as instructed in section 3.1.

8. Following instructions in the first two subsections of section 7, set up the testing hardware.

9. Configure AFT using the instructions from appendix A.

1.2 Testing automation USB-stick

In this document there are multiple references to a testing automation USB-stick. This stick is intended to contain programs and other tools required for maintaining a complete testing automation system.

Note: At the time of writing, the location is temporary and doesn't include pre-made Debian images due to their size.

The content can be downloaded from ta-usb-stick branch in the AFT git repository and copied to any suitably large USB-stick.

The content was bundled together during the creation of this document, in December 2015. Therefore the programs may be out of date.

1.3 SSH-keys

In the testing automation system SSH-keys are used extensively for authentication. A default SSH-key is included in the testing automation usb-key in folders `gigabyte` and `minnowboard`. Additionally, the SSH-key is added to the `root.ssh` folder in each pre-made Debian image. **It is highly recommended that these SSH-keys are replaced with your own ones!** Leaving the default keys in place may expose your system to attacks.

1.4 Foreword

This document was originally written over November and December 2015 to relay information regarding testing automation setup in an embedded Linux project. The level of detail is intentionally high to allow the next operator to understand all details of the system.

This document covers the configuration and construction instructions only for the testing automation part of the continuous integration system used in the project. The testing automation subsystem is not coupled to the build infrastructure other than by the image transmission.

In the first section the configuration and installation of the testing harness, the testing server, is covered. In the second and third sections the usage of the most important pieces of software are detailed. In section 5 the different kinds of power cutter devices are explained and their associated software is explained. In sections 6-9 the individual testing platforms are described. Finally, in the appendices, the AFT implementation details and privilege reduction options are explained.

2 Testing harness

The testing server is, in this document, called testing harness. The testing harness is responsible of fetching the images from the source and executing AFT to flash the images on various target devices. The system has been tested using *OpenSUSE 13.2*.

The testing harness acts to the testable devices as a network file system server, and as a DHCP-server. It is highly recommended that the devices are put in an isolated network.

Testable images can be copied from the builders using many means. Both NFS and standard IP transmission using `wget` have been tested and found reliable. Because the images are most likely copied over Ethernet, and the testable devices are in a separate network, the testing harness must have two network cards.

In the following section the configuration is detailed. The commands are expected to be run using `root` privileges. A fresh installation of OpenSUSE 13.2 with SSH-server, no desktop environment and working internet connectivity are assumed. Firewall configuration is not provided, it is recommended that the testing harness is only kept in a trusted network.

Configuration of AFT is explained in appendix A.

Base system

A set of common tools and utilities:

```
zypper install git gcc gcc-c++ make automake cmake autoconf man emacs nano screen tree attr dnsmasq gdisk nfs-kernel-server python
        python-setuptools python-devel python-tk python-idle ipython libusb-1_0-devel usbutils
```

OpenSUSE comes with `YaST` configuration utility. It can be used to configure most base OS options. Using `yast` configure:

1. The secondary Ethernet card to have static IP `192.168.30.1`, and subnet mask `255.255.255.0`
2. Create user `tester` and add it to `wheel` group

The secondary local area network is used for the flashing of PC-devices 3.1. The `tester`'s home will be set as shared folder over NFS for the `192.168.30.0/24` network. The `tester` is also the recommended user for automated testing.

Enable `wheel` group as passwordless sudoers. Using `visudo` add to the end of the `/etc/sudoers` file:

```
%wheel ALL=(ALL) NOPASSWD: ALL
```

DFU-util

Flashing Edison requires the installation of `dfu-util` program. Dfu-util takes care of the DFU-protocol and the flashing of each partition on the target device.

At the time of writing (24.11.2015) the official latest version of DFU-util does not have a working USB-tree support. Fortunately a patch has been made, and the installation is therefore straightforward.

```
git clone git://git.code.sf.net/p/dfu-util/dfu-util dfu-util-src
cd dfu-util-src
wget http://sourceforge.net/p/dfu-util/tickets/6/attachment/0001-Fix-reimplement-USBPATH-support.patch
git am 0001*
./autogen
./configure
make
make install
```

Clewarecontrol

Clewarecontrol utility is used to control the power cutters made by Cleware GmbH. At the time of writing, the latest clewarecontrol, version 4.1, was broken. Therefore the version 2.8 is recommended.

To install clewarecontrol commandline utility:

```
wget https://www.vanheusden.com/clewarecontrol/clewarecontrol-2.8.tgz
tar -xvf clewarecontrol-2.8.tgz
cd clewarecontrol-2.8
make
make install
```

Usage instructions are given in the section 5.

Dnsmasq

`Dnsmasq` is used as a DHCP-server for the PC-devices. It also provides a table with MAC-to-IP-address conversion. Dnsmasq is configured to provide IP-addresses to the `192.168.30.0/24` subnet, to all devices connected (through a switch) to the secondary Ethernet port.

Add the following lines to `/etc/dnsmasq.conf` file:

```
dhcp-range=192.168.30.2,192.168.30.254,10m
interface=p1p1
```

Note: replace the `p1p1` with the network interface the PC-devices are connected to.

To enable dnsmasq service:

```
systemctl start dnsmasq.service
systemctl enable dnsmasq.service
```

If at some point the DHCP-exchange has to be debugged, stop the Systemd-service and start dnsmasq manually using:

```
systemctl stop dnsmasq.service
dnsmasq --no-daemon
```

Network File System

Network file system (NFS) is used to share the images and other files used in flashing for PC-devices. By convention the whole `/home/tester` is shared.

To share the folder, add the following line to `/etc/exports`:

```
/home/tester 192.168.0.0/16(crossmnt,ro,root_squash,sync,no_subtree_check)
```

Then enable the Systemd-service:

```
systemctl start nfs-server.service
systemctl enable nfs-server.service
```

On some platforms, also `rpcbind.service` has to be activated.

Udev (optional)

It is recommended to set udev-rules to create symbolic links from a human-readable file name to ttyUSB-devices used for PEM (section 4), serial output recording (appendix A) and to the USB-relay powercutters used for Edisons. The USB-to-serial adapters are often unreliable and tend to occasionally reset, which causes a re-initialization by Linux kernel. This also triggers a (re-)creation of the ttyUSB-device created by udev.

The recommended way to set the rules is by adding rules to, for example, `/etc/udev/rules.d/99-persistent-ttyusb`. A rule can look for example like:

```
SUBSYSTEM=="tty", ENV{ID_PATH}=="pci-0000:00:1d.7-usb-0:1.1.3:1.0", SYMLINK+="gigabytearduino3"
```

The `SUBSYSTEM` is the udev subsystem which is used by the target device, the `SYMLINK` is the name of the symbolic link created under `/dev`, and the `ID_PATH` is used to filter the specific device. To retrieve the ID_PATH, plug in the device, find the correct `ttyUSBX` device using e.g. `dmesg` and execute:

```
udevadm info /dev/ttyUSBX
```

After setting the rules, udev must be refreshed using

```
udevadm trigger
```

Note: Setting the rules using this method allows unplugging and re-plugging USB-cables to the same port, or rebooting the testing harness without re-configuring AFT. Otherwise the ttyUSB-device may change!

3 AFT - Automated Flasher Tester

AFT is a Python tool to flash the testable image to various kinds of devices. AFT tries to guarantee that the device under test is in a state where:

- The device can be rebooted and the image under test starts again.
- The device under test can be accessed over SSH without a password.

To achieve the first requirement, AFT flashes the image to the primary boot media of the device under test, and possibly modifies the BIOS settings to use that as the boot media.

The second requirement is achieved by injecting a known public SSH-key to `<root home>/.ssh/authorized_keys`. This allows the owner of the corresponding private key to connect the device without a password. AFT may also modify the network settings of the testing harness to make sure the device is still accessible after a reboot.

Finally AFT starts the pre-configured tests by executing either its internal test cases or by starting a subprocess. The test cases are assumed not to modify the BIOS settings. For this AFT has an integrated test runner which executes test plans based on configuration.

Because flashing often requires multiple boots during the process, and the only guaranteed way to power off a device is to cut its power, different kinds of external power cutters are used.

The execution syntax for AFT is

```
aft <machinetype> <imagefile>
```

where, at the time of writing, the options for `machinetype` are `gigabyte`, `galileov2`, `minnowboardmax` and `edison`. The imagefile is the image to be tested.

Optionally, AFT can also record the serial console output to file by using `--record` argument, assuming the serial device has been configured properly. This allows some level of logging even if the device fails to boot.

Details to the configuration of AFT is given in the appendix A. In the same appendix there is a a general overview to the classes involved and implementation details.

3.1 PC-devices

PC-like devices are devices which have a network interface with a fixed MAC-address and usually have a some sort of BIOS/EFI menu.

PC-like devices are flashed with the aid of a *support image*, a Linux image which can boot the device under test. The support image is used to copy the target image over NFS to the primary boot media, and to add e.g. the SSH-key to it.

The BIOS settings are modified using PEM, an Arduino UNO device with a keyboard emulating firmware. See section 4 for further information.

The biggest hurdle with PC-devices is usually the creation of a support image. The distribution used on this guide is Debian 8.2 because it supports out of the box a wide range of architectures and is generally known to be compatible with many kinds of devices. In simple cases the support image creation is just a matter of installing the operating system to a USB stick. In worse cases, e.g. with Galileo Gen 2, it requires the creation of custom kernel with board specific *Board Support Package* (BSP).

Support images for all currently supported devices are provided on the testing automation USB-stick.

By using a support image to also modify the target image, AFT can be run without root privileges. `Mount`-command for modifying images and mount points always requires root privileges, but this is executed only on the support image. Unfortunately gadget-devices require the modification of the image on the testing harness, so this is not viable in the general case.

PC-devices usually use some kind of a power supply which is connected to the mains powerlines. Therefore a natural place for a power cutter is in between the mains and the PSU.

Another option would be to strip the positive line between the PSU and the device, and install a power cutter in between. This is considered a worse solution because it requires more customization of the hardware.

The PC-devices described in this document include Gigabyte 6, MinnowBoard MAX 7 and Galileo Gen 2 8.

A support image is provided on the Testing automation USB-stick for each platform in the folder `debian-images`. These can be copied to another USB-stick using e.g.

```
dd if=<USB-root>/debian-images/minnowUSB.image of=/dev/sdX bs=8M
```

where the `sdX` refers to the block device you want to copy the image to.

3.2 Gadget-devices

Gadget-like devices are devices which are closer to a mobile phone or a traditional embedded MCU or SoC. These are often flashed using the DFU-protocol.

There are several DFU programmer utilities and flashing a device completely may require separate flashing calls to flash each memory media on the device. Depending on the flashing utility, it may be possible to detect errors during flashing. These shouldn't cause a failure in testing but instead just initiate another attempt at flashing.

Depending on the device, switching power on and off can be considerably more difficult with a gadget-device. If the testable device has an internal battery, it most likely requires somewhat complicated custom hardware to power cycle it. With gadget-devices extra care must be taken in the prevention of floating ground issues. If the device is powered over USB, a powered USB-hub is almost certainly required.

The only gadget-device in this document is the Intel Edison 9, which uses dfu-util (2) for flashing and a USB-relay as a power cutter.

4 PEM - Peripheral EMulator

Note: The instructions work only on Arduino UNO!

Peripheral Emulator is a device created out of an Arduino UNO and USB-to-serial adapter for emulating a keyboard or another peripheral device. To be compatible with the PC 97 standard, and especially BIOS mode, the firmware supports 6-key-rollover.

PEM works by flashing both the Atmega 328U and Atmega 16U2 chips on the Arduino UNO R3 board.

The Atmega 328U is flashed with a firmware that takes care of the communication between the testing harness and the USB-to-serial adapter. The firmware receives keyboard states from the testing harness over serial data line and responds with an acknowledgement after each successful packet.

The Atmega 16U2 is converted from its original USB-to-serial mode to an actual 6KRO HID-emulator. The 16U2 receives the packets from the 328U, interprets them and sends them over the USB-interface using HID protocol.

PEM can be installed by issuing the following commands:

```
git clone https://github.com/01org-AutomatedFlasherTester/pem.git
cd pem
sudo python setup.py install
```

Note: PEM installation files are also included in the Testing automation USB-stick in the `installationfiles` folder.

4.1 Creation of a PEM-Arduino

The creation of a PEM-Arduino requires

- An Arduino UNO R3
- USB-to-serial adapter
- 3 wires to connect the USB-to-serial adapter to Arduino
- A jumper or a piece of wire
- USB A to USB B cable
- USB A to mini/microUSB (depending on the USB-to-serial adapter)

The programs required are the Arduino IDE 1.6: (`https://www.arduino.cc/en/Main/Software`) and dfu-programmer 0.62: (`https://dfu-programmer.github.io/`).

1. Plug the Arduino UNO to your computer using the USB-B to USB-A connector.

2. Start the Arduino IDE. Select the correct port in the `Tools->Port` menu. Open the `<PEM git repository>/src/pem_Arduino/pem_Arduino.ino`. Flash the sketch in the Arduino. Close the IDE.

3. With the Arduino connected to computer (and powered), connect briefly using a jumper or a piece of wire the two pins closest to the Arduino's USB-B port. This resets the Atmega 16U2. Now you should be able to get (scrambled) output by issuing:

   ```
   sudo dfu-programmer atmega16u2 dump
   ```

4. The firmware for the 16U2 is included in the testing automation USB-stick but it can also be downloaded from `http://hunt.net.nz/users/darran/weblog/b3029/Arduino_UNO_Keyboard_HID_version_03.html`. To flash the firmware, in the testing automation USB stick folder `arduinokb`, issue the following command:

   ```
   sudo sh flash.sh
   ```

   This script first erases the 16U2, then flashes the firmware and then resets the chip.

5. To verify that the system works, see the usage instructions in the following subsection.

4.2 Usage instructions

PEM has a graphical user interface for recording keyboard sequences. To make the keyboard sequence as reliable as possible, it is good idea to use constant intervals between the key presses. Additionally, most BIOS menus support e.g. PageUp and PageDown keys which move the cursor to the top or the bottom of the list (that is, to a known, fixed location).

When using the peripheral emulator, connect the USB-to-serial to your testing harness or recording computer using the mini/microUSB connector. Connect the USB-to-serial pins to corresponding Arduino pins: `TX => 10`, `RX => 11` and `GND => GND`.

When PEM is started, it doesn't start sending the key presses before the target device can detect them. This is achieved by waiting for the Arduino to boot up. Arduino is powered by the USB-B line, and therefore gets the power when the target device boots. Because the device under test power is controlled using a mains power cutter, in production, the PEM starts sending the key presses only exactly when the device boots.

To start the recording interface, use

```
sudo pem --interface serialconnection --record <target_file> --port </dev/ttyUSBX>
```

Note: replace the `/dev/ttyUSBX` with the `ttyUSB`-device corresponding to the USB-to-serial adapter, which can be found using e.g. `dmesg`.

In the recording interface, to start recording a sequence, press the `Start` button. After that, each key press sent to the PEM UI gets sent to the target device. To stop recording and to save the last recording to the file given in the command line arguments, press the `Stop` button.

To clean up the keyboard sequence, a LibreOffice Calc spreadsheet is provided on the testing automation USB-stick at `<USB-root>/arduinokb/sequence-editor.ods`.

To execute a keyboard sequence, use the following command

```
sudo pem --interface serialconnection --playback <playback_file> --port </dev/ttyUSBX>
```

5 Power cutters

Controlling the power on the target devices is always done using a power cutter in the power line. Power cutters are used because powering off cleanly is not reliable. In some PC-devices, the USB-ports also remain powered unless the power is completely cut off.

5.1 Cleware powercutters

In our setup we have used two kinds of power cutters: for PC-devices, power switches by Cleware GmbH are used. Cleware cutters are controlled over USB using a command line tool `clewarecontrol`. At the time of writing, the latest version of `clewarecontrol`, 4.1, is unstable at least on Linux Fedora 22. It is recommended to use the version 2.8.

The cutters can be listed by issuing

```
clewarecontrol -l
```

A switch can be turned on and off using the following command

```
clewarecontrol -d <device ID> -c 1 -as X [1|0]
```

where device ID is the ID of the cutter device, `X` is the index of power socket in the cutter (zero-based), and the last number is 1 for switching the socket on, and 0 for switching it off.

5.2 USB-relays

The Edison is controlled using a USB-relay in the +5V line on the USB-cable. The relay is controlled using a Python script shipped with the AFT. The script sends binary data over serial to switch the relay on and off.

The USB-relays can be ordered from e.g. eBay. In some cases the switches are incorrectly recognized by the testing harness. The serial controller identifies itself as something else but a USB-to-serial adapter. The cause is probably incorrect internal firmware.

This can be fixed by adding a custom USB device ID to the correct USB-to-serial driver. The driver depends on the USB-to-serial chip used in the relay, common ones being `cp210x` and `ftdi_sio`. Adding a new device can be done in a root terminal by using for example

```
echo "0b00 3070" > /sys/bus/usb-serial/drivers/cp210x/new_id
```

The device ID can be found with the help of e.g. `lsusb`. If the driver is not loaded, it can be loaded manually using `modprobe`.

6 Gigabyte GB-BXBT-3825

Gigabyte is an IoT-gateway platform. It has a Gigabit ethernet, two USB 2.0 ports and one USB 3.0 port, HDMI- and VGA-outputs and internal WLAN and Bluetooth cards. It comes with 2 GB of ram and Intel Atom E3825 - the same one as in MinnowBoard MAX. It also has a 500 GB internal hard drive, which is the boot device used for testing.

The BIOS of Gigabyte has basic settings for boot option priorities and overriding the boot priorities. It also has options to restart the device on AC power loss, which has to be set as the booting is executed using a powercutter. Gigabyte supports Secure boot, but also has options for disabling it and to let users add their own keys. To enter the BIOS press delete-key during boot.

Note: the BIOS only supports one boot media with legacy boot (MBR), which must be set as the `Boot Option #1` in `Boot => Hard Drive BBS Priorities`.

Note: the USB 3.0 port in the front of the device doesn't work as a boot device if used with a USB 3.0 stick.

Note: Gigabyte BIOS is notoriously buggy. It sometimes creates invisible options to the boot override menu which replace a valid option but don't work. To fix this, reflash the BIOS.

Note: At the time of writing, Gigabyte requires a screen attached to the HDMI port in order to boot in MBR mode. A screen in the VGA port does not work.

Gigabyte supports only 12V DC input and requires a 2.5 A powersupply. The DC-plug has a positive center.

6.1 CI-integration

Gigabyte is a PC-like device, similar to Galileo Gen 2 and MinnowBoard MAX. A Debian support image is created on a USB-stick. It is recommended that the support image is created with a legacy (non-EFI) bootloader. The BIOS settings are controlled with PEM-Arduino. The internal hard drive is used for the target image.

The internal ethernet adapter is used for networking.

The PEM keyboard sequence should select the support image from `Save & Exit => Boot Override` menu. The image boot options should (automatically) be the top options in the boot option priorities so the keyboard sequence for testing should be empty.

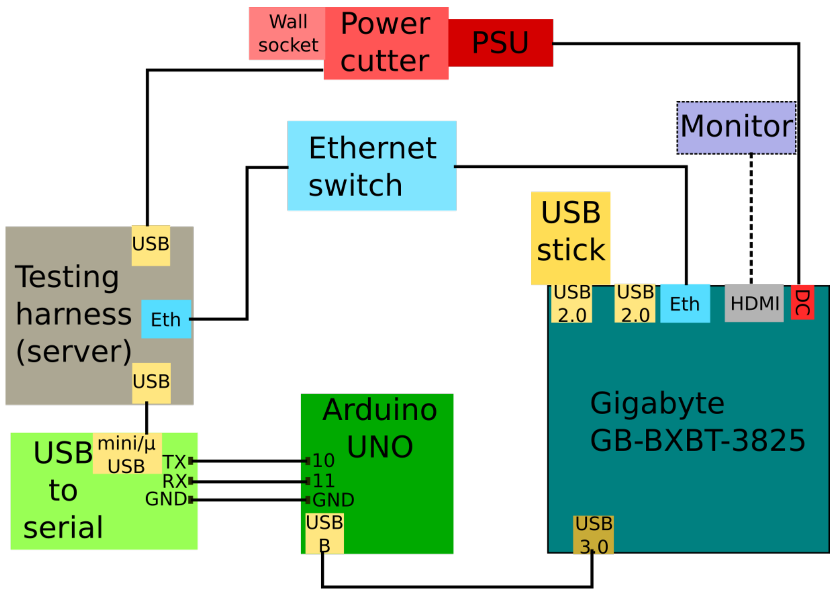


Figure 2: Gigabyte GB-BXBT-3825 wiring diagram

6.2 Debian on Gigabyte

These instructions are for creating a bootable Debian USB-stick for Gigabyte with MBR and persistent storage.

1. Make sure the internal hard drive doesn't have a bootloader, or nuke the hard drive:
2. Update the BIOS

```
dd if=/dev/zero of=/dev/sda bs=8M count=500
```

3. In the BIOS disable Secure boot and set the

   `Chipset => Restore AC Power Loss to Power On`.
4. Dd a 64-bit Debian installation DVD to USB-stick:

```
dd if=debian-8.2.0-amd64-DVD-1.iso of=/dev/sdX bs=8M; sync
```

5. Plug the USB-stick on a USB 2.0 port, boot the device and enter BIOS. **Note:** if the USB-stick is a USB 3.0 stick, it will not boot from the USB 3.0 port!
6. Select the USB-stick as the primary legacy boot option in `Boot => Hard Drive BBS Priorities`. Select the USB-stick as the boot override option in `Save & Exit => Boot Override`.
7. Only plug the target USB-stick after Debian installer menu is visible.
8. Start Debian installation.
9. When prompted for missing firmware files, don't try to install them at this phase (answer 'no').
10. Select hostname, use e.g. Debian-gigabyte.

11. Root password: `rootme`
12. User name: `user`, real name: `user`, password: `user`.
13. For partitioning, automatical partition for entire disk with all files in one partition works. Remember to select the correct USB-drive.
14. When prompted for software selection, the spacebar enables and disables options. Disable Debian desktop environment and print server, but enable SSH server. Enter to continue.
15. Once the installation is finished, remove the installation media, set the remaining USB-stick as the primary legacy boot option and boot the newly installed Debian.
16. Once booted, log in as `root`, plug in the test automation USB-stick and execute:

```
mkdir temp; mount /dev/sdc1 temp; cd temp/gigabyte; ./installgigabyte; cd; umount temp; rm -r temp
```

    See the comments on the test automation USB-stick `gigabyte/installgigabyte` script for details of the post installation configuration.

17. Reboot the device and from the BIOS menu, boot the Debian image.
18. Manually flash the internal hard drive with a current testable image.
19. Reboot the device, adjust the boot priorities so that the primary options are from the testable image.

7 MinnowBoard MAX

MinnowBoard MAX is an open development board with a 64-bit Intel Atom E3825 CPU (the same as in Gigabyte) and 2 GB ram. It provides one microSD slot, a SATA2 connector, one of each USB 2.0 and USB 3.0 sockets, gigabit ethernet and a 6-pin serial output. It also has a microHDMI connector for monitors.

MinnowBoard MAX uses EFI for hardware initialization. The BIOS menu can be entered by pressing `F2` during boot. The boot priorities can be overridden in the `Boot Manager` menu, and the priorities can be adjusted in `Boot Maintenance Manager`.

7.1 CI-integration

MinnowBoard MAX is a PC-like device, similar to Gigabyte and Galileo Gen 2. A Debian support image is created on a USB-stick. For MinnowBoard MAX Debian, an EFI-enabled bootloader is used but it has to be slightly modified for more convenient and stable test automation. The instructions for this are at the end of section 7.3.

The internal ethernet adapter is used for network connectivity.

After the bootloader modification, the PEM keyboard sequence can be used to select `EFI USB Device` as the boot device in the `Boot Manager` menu. The boot partitions of the image under test should be set as the primary boot options in the `Boot Maintenance Manager` so that an empty keyboard sequence can be used to boot it.

The 6-pin serial output can be used for boot console recording at 115200 bauds.

The microSD-card is used for the target image with MinnowBoard MAX.

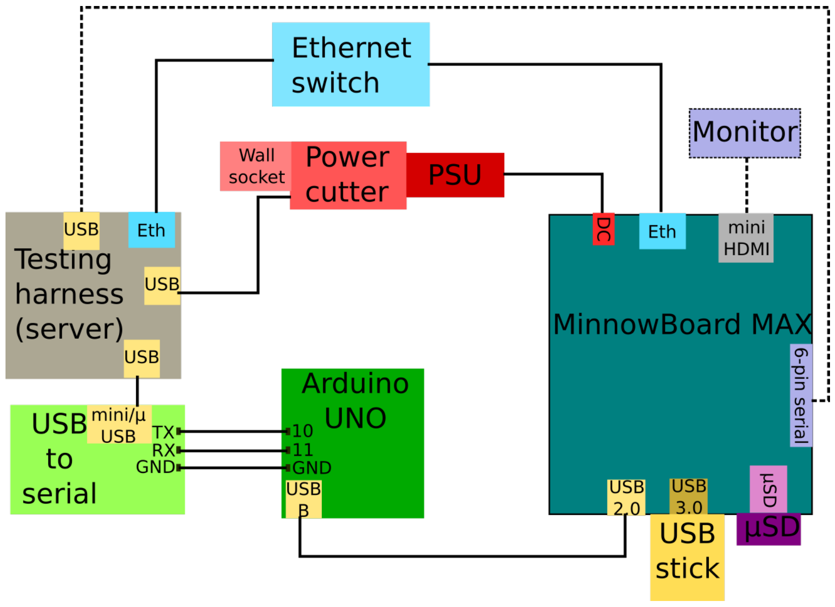


Figure 3: MinnowBoard MAX wiring diagram

7.2 Flashing the BIOS on MinnowBoard MAX

MinnowBoard MAX supports both 32- and 64-bit firmwares. The official firmware site is `https://firmware.intel.com/projects/minnowboard-max` but the firmware version `0.83` has been added to the testing automation USB-stick with both 32- and 64-bit flashing utilities. The following instructions explain how to flash the firmware using the testing automation USB-stick.

1. Attach the testing automation USB-stick to MinnowBoard MAX along with a keyboard and a monitor.
2. Reboot the device and enter the BIOS menu. Determine the architecture of current BIOS from the third line in the header. For example, *MNW2MAX1.**X64**.0083.R01.1509181113*.
3. Select `EFI Internal Shell` as the boot media. (`Boot Manager => EFI Internal Shell`).
4. Enter the USB-stick file system with the help of `Mapping Table`. In this example the USB-stick has been mapped to `FS0`. Note that the keyboard layout is US.

```
FS0:
```

5. Change to the `minnowboardbios` folder:

```
cd minnowboardbios
```

6. Based on the BIOS version architecture, execute the correct updater; for X64 system `MinnowBoard.MAX.FirmwareUpdateX64.efi` and for IA32 systems the IA32 version.

```
MinnowBoard.MAX.FirmwareUpdateX64.efi MinnowBoard.MAX.X64.083.R01.bin
```

7. The system will shut down after flashing is complete. You will have to reconfigure the BIOS settings after the update.

7.3 Debian on MinnowBoard MAX

Because MinnowBoard MAX has only two USB-ports, and one of them is required for a keyboard, a microSD-card must be used for installation media. The BIOS doesn't support USB hubs but they can be used after the OS is booted.

1. Flash a 64-bit Debian installation image on a microSD-card:

   ```
   dd if=debian-8.2.0-amd64-DVD-1.iso of=/dev/sdX bs=8M; sync
   ```

   Select `X` to match the SD-card device on your machine with the help of e.g. `lsblk`.

2. Using e.g. `gparted` remove the default partitioning of the target USB-stick.
3. Insert the SD-card to a MinnowBoard MAX. Do *not* insert the USB stick yet!. Reboot the device and from BIOS menu select the Misc device to boot from the SD-card. A Debian installer menu should appear.
4. Insert the target USB-stick.
5. Start installation, select language (English), locale (Irish) and keyboard layout (Finnish).
6. When prompted for CD-ROM drivers, select `No`. Then manually select the CD-ROM module and device (`yes`), select the module needed for access the CD-ROM (`none`) and device file for accessing the CD-ROM: `/dev/mmcblk0`.
7. When prompted for missing firmware files, don't try to install them at this point (answer: No).
8. Select hostname, use e.g. Debian-MinnowMAX, and leave the network name empty.
9. Set the root password, use `rootme`
10. Create a user, use user name: `user`, real name: `user` and password `user`.
11. Adding a network package source is not necessary, it is added by a post-install script.
12. When prompted for software selection, the spacebar enables and disables options. Disable Debian desktop environment and print server, but enable SSH server. Enter to continue.
13. After the installation is finished, eject the microSD-card, select `debian` as the boot device in the BIOS `Boot Manager` and login to the Debian.
14. Format the microSD-card and from the testing automation USB-stick, copy the contents of the contents of `minnowboard` folder to the card.
15. Plug the SD-card in the MinnowBoard MAX, execute the following:

    ```
    cd; mkdir temp; mount /dev/mmcblk0p1 templs; cd temp; chmod +x installminnow; ./installminnow; cd; umount /dev/
            mmcblk0p1; rm -r temp
    ```

    See the comments on the test automation USB-stick `minnowboard/installminnow` script for details of the post installation configuration.

16. Reboot the device using `reboot` and from BIOS menu select the EFI USB Device as the boot device.
17. manually flash the microSD-card with a current testable image.

18. Reboot the device and enter BIOS menu.

19. In `Boot Maintenance Manager` delete the `debian` boot entry. Then adjust the boot option order so that the previously flashed image is the primary boot device.

8 Galileo Gen 2

Galileo Gen 2 is a Quark development board with 256 MB of ram. It has integrated 100 ethernet connectivity, a microSD slot, a USB 2.0-port and a 6-pin serial connection socket. Underneath the board there is a full-size Mini PCIE card slot. Therefore if a half-size Mini PCIE card (e.g. Intel Wireless N 7260) is used in testing, a half- to full-size adapter has to be used.

Galileo Gen 2 supports input voltage between 7V and 15V. The included power supply is 12V, 1.25A. The DC-plug has a positive center.

8.1 BIOS

The BIOS menu of Galileo Gen 2 is very limited. It only provides an option to select a boot-device by pressing `F7` during boot time. The relevant options from CI-perspective are the microSD-card, represented as "Misc device" in the boot menu, and the USB-stick, represented with the make and model of the stick in the boot menu. Other options include e.g. EFI-shell and the yocto-image flashed to the internal storage.

The primary boot option is the "first misc or USB-device", which is not specified further. The first boot option depends on the order in which the microSD-card and the USB-stick have been inserted. If the test cases include a test which reboots the device, the testable image has to be put in the primary boot option, and the support image has to be on the secondary boot device.

8.2 CI-integration

In CI-system Galileo Gen 2 is a "PC-like" device, and therefore similar to Gigabyte and MinnowBoard MAX. To integrate Galileo Gen 2 in automated testing, the USB-socket has to be extended using a USB-hub, to connect both a PEM-Arduino and a USB-stick with bootable support image. The microSD-card is used for the target image.

The integrated ethernet adapter is used for network connectivity.

The 6-pin serial output can be used for boot console recording at 115200 bauds.

The PEM keyboard sequence to boot the support image should be the selection of secondary boot device, while the sequence to boot the testable image should be empty.

8.3 Debian on Galileo Gen 2

The following instructions are based on the preliminary work by Igor Stoppa in creating a bootable SD-card Debian image with persistent storage for the first generation Galileo. Several modifications and additions had to be made to his instructions but the basic idea has remained the same. The main difficulty is finding a Linux distribution which is bootable with an i586 device, as almost all "32-bit" distributions are using at least i686 architecture. Additionally, the kernel configuration required for Galileo Gen 2 had to be searched experimentally, to get

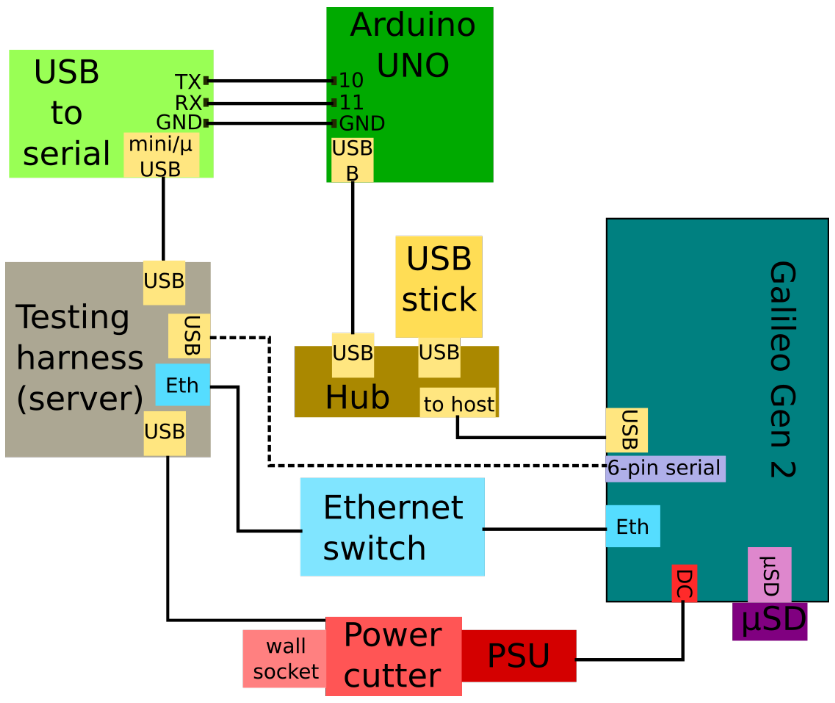


Figure 4: Galileo Gen 2 wiring diagram

all the features required for AFT-integration. The end result is an image which can be booted from both an SD-card or a USB-stick.

The high-level workflow is as follows:

1. Prepare a 32-bit Debian Virtualbox virtual machine
2. Build a custom Linux kernel using BSP provided by Intel
3. Using debootstrap, prepare the base OS
4. Add other required programs and configuration to the image.
5. Manually construct the boot partition using the custom kernel and EFI GRUB from a Galileo Gen2 image provided by Intel.

The instructions have been tested on 18.11.2015 using a Fedora 22 host machine, 32-bit Debian 8.2 and VirtualBox 5.0.10.

In the following subsections the instructions are described in more detailed manner.

Virtualmachine creation

1. Using VirtualBox, create a new Debian (32 bit) virtualmachine with 2 GB RAM and 50 GB dynamically allocated VDI harddrive. **Note:** You have to enable a NAT network adapter in the VM settings to have an internet connection inside the VM guest.

   It is recommended to enable also bidirectional shared clipboard and drag'n'drop support.
2. Add a 32-bit Debian installation media in the virtual CD-drive of the VM, start the VM and install it following the installer instructions.

3. Remove the installation media from the virtual CD-drive and restart the VM.
4. Once booted, start a terminal and `su` to root terminal. **Note:** If you are behind a proxy, it is recommended to set the settings at this point in `.bashrc`.
5. Comment out CD/DVD-entries in

```
vi /etc/apt/sources.list
```

6. Install required packages to the Debian system:

```
apt-get install debootstrap vim build-essential binutils git gawk chrpath kernel-package fakeroot libncurses5-dev gparted dkms
```

   When prompted about the version of kernel config, it is recommended to use the package maintainer's.

7. Install VirtualBox guest additions to the guest and the host systems using instructions from VirtualBox manual.
8. Create a shared folder between the host and the VM. In these instructions the folder is named `shared` on both the host, and the guest machine, under the root home.
9. Download and unpack the SDCard tarball from `https://downloadcenter.intel.com/download/24355/Intel-Arduino-IDE-1-6-0` . At the time of writing, the file was named `SDCard.1.0.4.tar.bz2`
10. copy the extracted `image-full-galileo`-folder to the shared folder. The folder should contain at least `grub.efi`, and `boot/grub/grub.conf`.

Kernel construction

1. Enter the virtualmachine, start a terminal and switch to root shell using `su`
2. Download Intel Quark Board Support Package (BSP) sources from `https://downloadcenter.intel.com/download/23197/Intel-Quark-BSP` using eg. `wget` .

   Note: At the time of writing the latest version (v. 1.20) had a non-existing base commit ID. Therefore an older version (eg. 1.10) must be used.

3. Unpack the BSP using

```
7za e *.7z; tar xvf quark_linux_*.tar.gz -C bsp
```

4. Git clone the latest stable kernel repository

   `git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`

5. Checkout the commit SHA found inside `<bsp>/quark_linux_v3.8.7+v1.1.0/upstream.cfg`

```
git checkout 531ec28f9f26f78797124b9efcf2138b8974a1e
```

6. Apply the patches required for Galileo Gen2:

```
git am ../bsp/quark_linux_v3.8.7+v1.1.0/*.patch
```

7. Copy the provided kernel config from the stick `Galileov2/.config` to the kernel repository root.

8. If you wish to customize the kernel configuration, use `make menuconfig`. For example, adding all the USB-drivers may be desirable.

9. Compile kernel:

```
fakeroot make-kpkg --initrd kernel_image modules_image}
```

   The .deb package should appear on one folder *above* your working directory.

Debian base construction

1. Create the empty image file and start partitioning

```
dd of=galileoimage.img bs=1 count=0 seek=2G; losetup -f galileoimage.img; gparted /dev/loop0
```

2. Using gparted, first add a partition table (msdos is fine) and then add 3 partitions: fat32 100MB, linux-swap 500MB, and ext4 the rest. Finally, add `boot` and `esp` flags to the fat32 partition.

3. Mount the partitions

```
mkdir usb_boot; mkdir usb_root; mount /dev/loop0p1 usb_boot; mount /dev/loop0p3 usb_root
```

4. Prepare the base system using debootstrap:

```
debootstrap --arch i386 stable usb_root http://http.debian.net/debian
```

5. Rest of the preparation of the chroot environment:

```
mount --bind /dev usb_root/dev; mount --bind /dev/pts usb_root/dev/pts; cp linux-image*.deb usb_root/root/
```

Image configuration

1. In root shell home, copy the `authorized_keys`-file from the usb-stick to the chroot root home:

```
cp <authorized_keys> usb_root/root/authorized_keys
```

2. Start the chroot environment:

```
chroot usb_root
```

3. Install packages required for maintenance:

```
apt-get install locales ntp openssh-server vim initramfs-tools net-tools bash-completion python python-pip nfs-common nfs-
        server nano git bmap-tools parted attr gdisk tree
```

4. Add the serial terminal:

```
echo "t0:2345:respawn/sbin/getty -L 115200 ttyS1 vt102" > /etc/inittab
```

   Note: : The terminal interface (`ttyS1`) depends on the BSP version. This has been tested for 1.10 but an earlier version may use `ttyQRK1`.

5. Set the password for root user: `passwd`

   Note: : The root password in support images so far has been `rootme`

6. Set the hostname

```
echo "Debian-Galileov2" > /etc/hostname
```

7. Add the hostname to name resolution in `etc/hosts`:

```
127.0.0.1 localhost Debian-Galileov2
::1  localhost ip6-localhost ip6-loopback Debian-Galileov2
```

8. Configure network interface(s) `vim /etc/network/interfaces`:

```
auto eth0
iface eth0 inet dhcp
```

   Note: It is a good idea to add multiple network interfaces (`eth1, eth2, ...`) if you want to use the same stick in multiple devices.

9. Permit root login over ssh:

```
echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
```

10. Add ssh-key:

```
mkdir -p /root/.ssh; chmod 700 /root/.ssh; mv /root/authorized_keys /root/.ssh/; chmod 600 /root/.ssh/authorized_keys
```

11. Create NFS and testable image mount point directories:

```
mkdir /mnt/img_data_nfs; mkdir /mnt/super_target_root; mkdir /mnt/target_root
```

12. Add entries to `/etc/fstab`: `vim /etc/fstab`:

```
/dev/sda3 / ext4 defaults 0 0
192.168.30.1:/home/tester /mnt/img_data_nfs nfs rsize=8192,wsize=8192,timeo=14,intr,nolock,auto
```

13. Install the kernel package:

```
dpkg -i /root/*.deb
```

14. Fix `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=738575`. The command replaces every `lock` instruction with a `nop`. This works because Quark is a single-core processor:

```
for i in ‘/usr/bin/find /lib -type f -name *pthread*so‘; do cp \$\{i\} \$\{i\}.bak; sed -i "s/\bl xf0\bl x0f\bl xb1\bl x8b/\bl
        x90\bl x0f\bl xb1\bl x8b/g" \$\{i\}; done
```

15. Leave the chroot environment using `ctrl+d`

16. Copy the EFI files from shared folder:

```
mkdir -p usb_boot/EFI/BOOT; \linebreak cp shared/image-full-galileo/grub.efi usb_boot/EFI/BOOT/bootia32.efi; \linebreak
        cp -a shared/image-full-galileo/boot usb_boot
```

17. Copy the kernel files to boot partition:

```
cp usb_root/boot/* usb_boot
```

18. Modify the `usb_boot/boot/grub/grub.conf`, make the following entry and remove the others:

```
title Debian
    root (hd0,0)
    kernel /vmlinuz-3.8.7+ root=/dev/sda3 3 console=ttyS1,115200n8 earlycon=uart8250,mmio32,
          $EARLY_CON_ADDR_REPLACE,115200n8 vmalloc=3844M reboot=efi,warm apic=debug rw LABEL=boot
          debugshell=5
    initrd /initrd.img-3.8.7+
```

Note: You may have to modify the `vmlinuz-3.8.7+` and `initrd.img-3.8.7` to correspond your kernel and initrd files.

Note: If you want to boot the image from a SD-card, modify the `root=/dev/sda3` option to point to the SD-card: `root=/dev/mmcblk0p3`.

19. Unmount everything and copy the image to the shared folder.

```
umount /dev/loop0p1; umount /dev/loop0p3; cp galileoimage.img shared
```

20. Poweroff the VM.

21. Inside the host system, dd the `galileoimage.img` to a USB-stick:

```
sudo dd if=shared/galileoimage.img of=/dev/sdX bs=8M
```

Note: modify the X to correspond your USB-stick device which can be found using eg. `lsblk`.

9 Intel Edison

Intel Edison is a development platform intended for wearable devices. It is a SoC with two Intel Atom cores and one Intel Quark core, GB of internal RAM and integrated Bluetooth and Wi-Fi. The Atoms are normal x64 cores while the Quark is roughly an i586 processor.

Edison can be mounted on an Arduino development kit which provides Arduino UNO compatible pin layout. The development board also has a USB-serial interface, a mechanical switch to select between a micro-USB device controller and a normal USB socket, a DC plug and a microSD socket.

Edison uses a hacked U-Boot for hardware initialization.

From flashing perspective, Edison is closer to a mobile phone or similar "gadget" device than a complete PC. The OS image is written using DFU.

9.1 USB-interfaces

When an Edison is plugged to a computer, it is first detected as an Intel Merrifield device (`8086:e005`), which is used to recover the DFU-utility in the firmware. After a few seconds the device disconnects and reconnects as DFU-device, Intel USB download gadget (`8087:0a99`). At this stage the OS can be flashed. If DFU-communication is not initialized in a couple of seconds, the device disconnects again and finally boots normally.

After the boot process, the device is detected with multiple interfaces with ID `8087:0a9e`. It can be used as a USB-storage, USB-serial device (at `/dev/ttyACM`*`X`*) and as a USB-network interface.

The USB-network interface is used for network connectivity. This interface has to be configured on both the image under test and the testing harness. Network configuration is handled by AFT 3.

The USB-network interface of the Edison gets a random MAC-address every time the image is flashed. This also causes the network interface on the testing harness being recreated and renamed.

If there are multiple Edisons attached to the testing harness, the only way to differentiate between them is the USB-tree or USB-paths. Using the paths, each physical USB-port can be differentiated from each other.

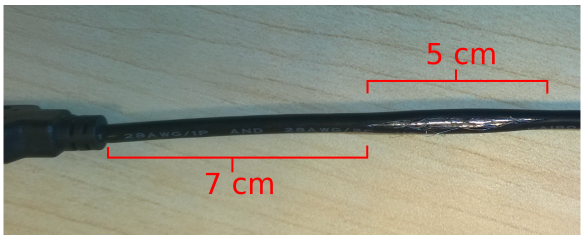
9.2 CI-integration

Integrating Edisons to the testing automation requires a way to power off and on the device programmatically.

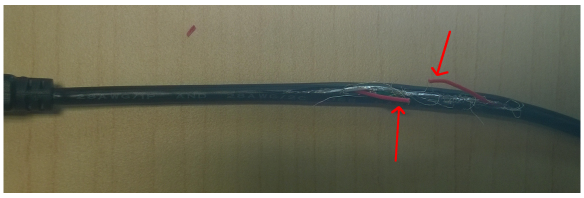
The best solution would be to find USB-hubs that supports Linux kernel's USB power options at `/sys/bus/usb/devices/usb1/power/` on per-port accuracy. Unfortunately this kind of USB-hubs are extremely difficult to find.

An implemented alternative is to use a power cutter on USB-cable's +5V line. Because the device is powered by USB, the power cutter also shuts down the device. This requires exposing the +5V cable, stripping it, and attaching the newly open ends to a USB-controlled relay, as instructed in figure 5.

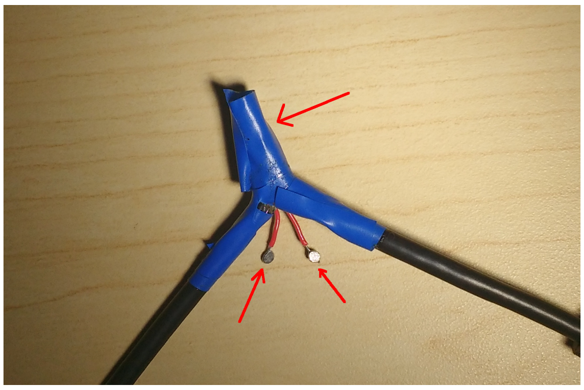
The USB-serial interface can be used for boot console recording at 115200 bauds.



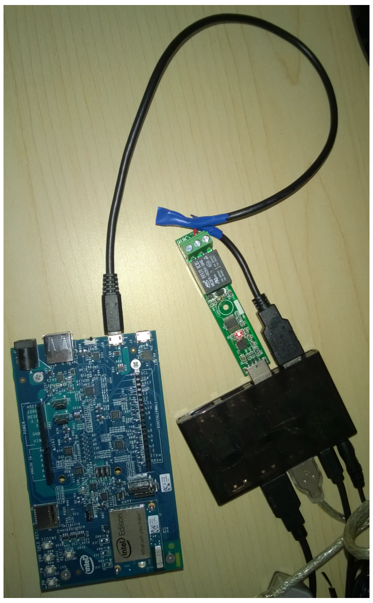
1. Expose the wire shielding by cutting a 5 cm slice through the jacket, about 7 cm from the USB-A plug.



2. Strip the shielding carefully. Expose the red wire (+5V) and cut it in half.



3. Strip the wire, solder small beads of tin on the tips. Cover the cut with electrical tape.



4. Final setup.

Figure 5: USB-power cutter cable.

A AFT implementation details

In this appendix, configuration options are detailed. Also, a high level description of most important AFT implementation decisions are explained.

A.1 Configuration

All AFT configuration is done with configuration files stored in `/etc/aft`.

Configuration files for the testable devices are stored in the subfolder `devices`. The configuration of a single physical device is combined out of these files.

In the `test_plan` folder is the configuration for each AFT test-plan.

devices/platform.cfg

The `platform.cfg` is the highest level configuration file. It is intended to store settings which are shared between all high-level device-types, ie. PC-devices or gadget-devices.

This is also the place where settings for automatic construction of the device topology can be stored. For example the `leases_file_name` refers to the location of the leases file used by `dnsmasq`. This file can be used to detect devices attached to the local network in which the PC-devices are kept in.

devices/catalog.cfg

The `catalog.cfg` is the configuration file describing each device type. These are the options shared by all devices of the same type.

Each section in `catalog.cfg` must include at least the `platform`, `cutter_type` and `test_plan` options.

The `platform` option is used to load the correct high-level device configuration from the `platform.cfg` file.

The `cutter_type` is used to determine the type of cutter used for the devices. At the time of writing the options are `clewarecutter` and `usbrelay`.

The `test_plan` option is the name of the name of the test plan configuration file under `test_plan` folder.

For *PC-devices*, the additional options are as follows:

- `target_device`: The block device the image is flashed to.
- `root_partition`: The partition to which the root of the image ends up after flashing the image. **Note:** this option is only used if there is no disk layout configuration file in the AFT invocation folder.
- `disk_layout_file`: The partition layout file name. Expected to be found in the working directory of AFT invocation.
- `service_mode`: A pattern which can be used to verify that the device is in service mode, that is, ready to be flashed. Compared against `/proc/version`.
- `test_mode`: Same as above but for the testing mode, that is, what is seen after the testable image has booted.

- `service_mode_keystrokes`: The keyboard sequence which switches the BIOS options to service mode.
- `test_mode_keystrokes`: Same as above but for testing mode.

For *Edison* devices there are no extra device specific options.

For *serial recording*, the add option:

- `serial_bauds`: The baudrate of the serial connection.

devices/topology.cfg

The `topology.cfg` file contains individual physical device specific information. The options you have to specify here depend on the device type and the power cutter used with it.

The mandatory options for all devices are the `model` and `id`. The information required to construct a cutter instance specific to the device is also almost certainly required.

The `model` is the device model. This is used to determine which device type from `catalog.cfg` is associated with the physical device.

The `id` is a unique identifier which is used as the name of the lock file associated with the device, when the device is in use. For PC-devices this should be the MAC-address. For Edisons this can be anything that is unique.

The options related to *Clewarecutters* are as follows:

- `cutter`: The ID of the cutter. This is taped on each cutter but can be also found using the `clewarecontrol` tool
- `channel`: The power socket of the cutter specified with `cutter`.

For *usbrelays* the only required option is `cutter`. This specifies the ttyUSB device associated with the cutter.

For *PC-devices* the mandatory options are as follows:

- `pem_interface`: the interface used with PEM. The only option at the time of writing is serialconnection.
- `pem_port`: The ttyUSB device for the USB-to-serial adapter connected to the PEM-Arduino.

For *Edisons* the mandatory options are as follows:

- `edison_usb_port`: The USB-bus and -port to which the USB-cable to Edison is attached. **Note:** Not the usbrelay but the cable.
- `network_subnet`: A *.*.*.*/30 subnet dedicated for this specific Edison device.

For *serial recording* the mandatory additional options are:

- `serial_port`: The ttyUSB device attached to the serial port of the target device.

aft.cfg

The `aft.cfg` is the global configuration file used to specify settings that affect the behaviour of AFT itself.

The `lock_file` option is the directory into which lock files can be created by users in *lock* group. For example on OpenSUSE this is `/var/lock`, while on Fedora it is `/var/lock/lockdev`.

The `serial_log_name` is the filename to which serial output is recorded under the AFT working directory.

The `aft_log_name` is the filename to which internal AFT log messages are stored.

The `nfs_folder` is the folder which is exported using NFS, and visible to the devices under test.

test_plan

The `test_plan` folder contains configuration for each test plan. In a configuration file each section define one AFT test case with the parameter `test_case` and the settings for that test. The `test_case` is associated with the correct test class by the *testcasefactory*.

A.2 Classes and files

AFT is aimed to be as easy to deploy as reasonably possible. Because it is also intended to be flexible and suitable for other projects, the code is also kept as simple and short as possible.

setup.py

The installation module. This file is responsible of deploying all AFT-related items, creating entry points and deploying example configuration files if they don't exist already.

device.py

Device is an abstract base class to define an interface for all device types. It requires the implementation of `write_image` and `get_ip` methods. The `write_image` should execute all the steps required for flashing the image. The `get_ip` should return an IP-address that is guaranteed to work for SSH-connection on the device instance.

In addition, the `test` method should set the device and host to be ready to execute the testing, and then `return test_case.run(self)`. This is the visited method in a visitor pattern.

The `record_serial` method is in the *device* class because the serial port used for recording is device-specific.

cutter.py

Cutter is an abstract base class to define an interface for all power cutters. It requires the implementation of `connect` and `disconnect` methods, to power on and off a power cutter.

devicefactory.py and testcasefactory.py

Factory modules to construct devices and testcases. These are the only locations where there should be string-to-Python-class conversion.

This is preferred over a perhaps more elegant `setuptools entrypoint` method for the addition of modules to AFT to keep the codebase simpler. The entrypoint methodology requires somewhat complicated installation method in the extension modules.

testcase.py

TestCase is an abstract base class to define an interface for all AFT test cases. These can for example call an external testrunner with options or run a simple test case themself.

config.py

A module used for parsing global configuration file `/etc/aft/aft.cfg`. The values are set as module attributes so that they can be referred using *aft.config.OPTION* syntax. Also sets sensible default values.

main.py

The entry point to AFT. The high-level execution sequence is

1. Construct all devices of the requested type
2. Reserve a device for this specific execution
3. Prepare the AFT test runner
4. Flash the image
5. Execute the test runner and run the tests

tools

General tools and subprocesses for AFT. This provides for example a safe subprocess execution call with timeout for both the testing harness and device under test.

default_config

Default example settings for the configuration files. Installed if they don't exist under `/etc/aft`.

devices

The device modules and their associated topology-generation modules.

cutters

Power cutter modules.

B AFT without root privileges

If none of the testable devices require actions which require root privileges, e.g. `mount` on the testing harness, AFT can be run without root privileges. In other words, if only PC-devices are used in the testing system, AFT can be run with lowered privileges.

The privileges that are mandatory for AFT are access to serial devices and the ability to lock files. The corresponding user groups on OpenSUSE are `dialout` and `lock`. These are easiest to set using `yast` on the testing harness and setting the user used for testing to these groups.

Additionally `clewarecontrol` always requires root privileges. By setting the `setuid` bit on the executable makes the program always run on its *owner user*. Because the programs under `/usr/bin` are owned by root, this provides the necessary privileges for those executions. The setuid bit can be set by issuing as a root user:

```
chmod 4755 /usr/bin/clewarecontrol
```