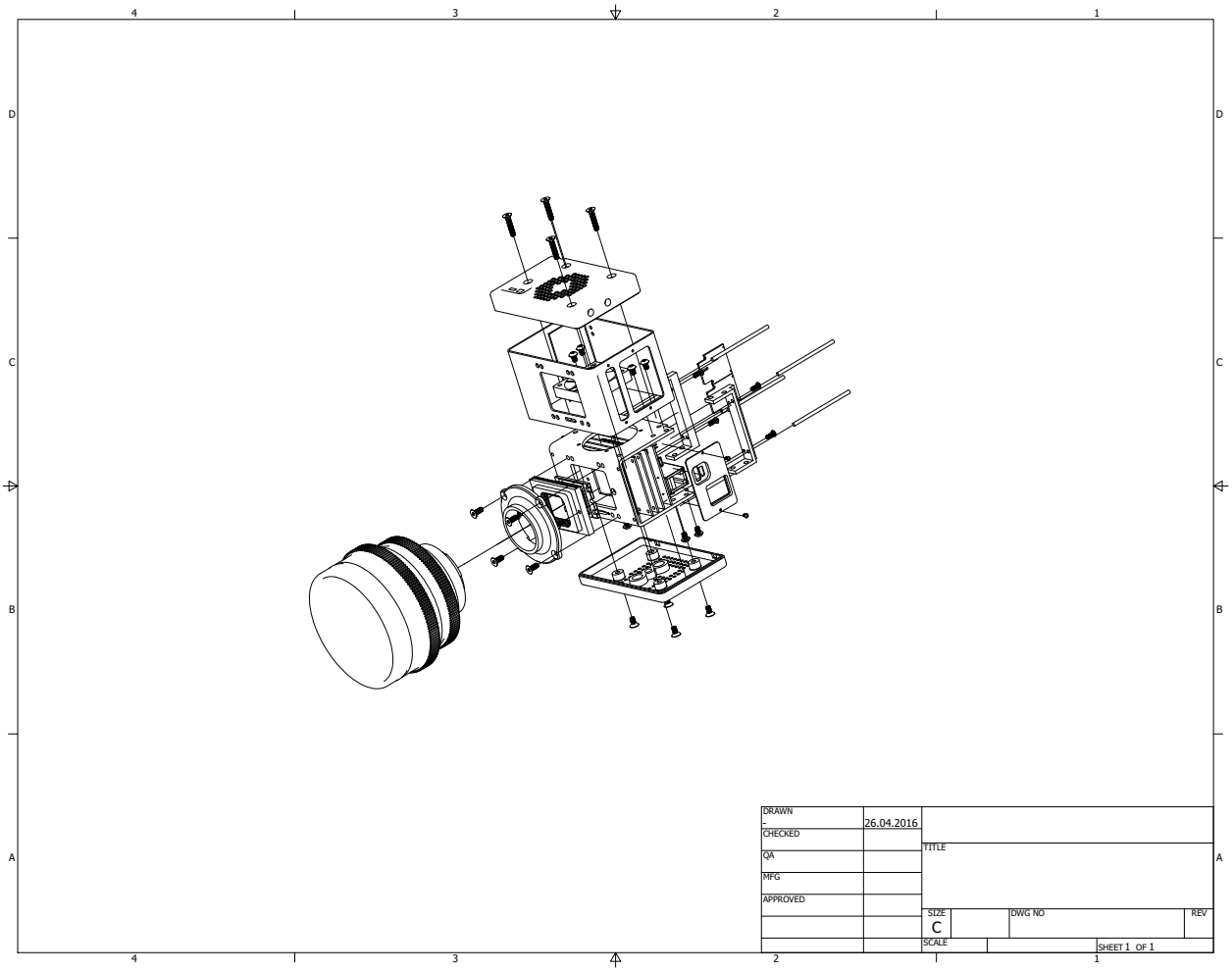
apertus
open source cinema



AXIOM Beta User Guide

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution 4.0 International Public License (CC BY-SA 4.0).

Published by apertus Association.

For more information, email team@apertus.org or visit apertus.org

Contents

Note: In some instances the instructions we have prepared are written in a manor that can be followed by people without a deep technical knowledge. If you are an advanced user please keep this in mind.

This document has been compiled using LaTeX and its files are stored in the apertus GitHub repository here. If you choose to make any changes to this document please notify a member of the Team so that those changes can be implimented into instances of this guide that are being made available in other locations and formats eg. the project's Wiki page.

1 General Informaion

Notes on Userspace: Arch Linux comes with systemd, which has one advantage that the boot process is incredibly fast. Standard tools such as sshd and dhcpcd have been preinstalled.

One idea to store camera relevant parameters inside the camera and provide access from most programming languages is to use a database like http://en.wikipedia.org/wiki/Berkeley_DB

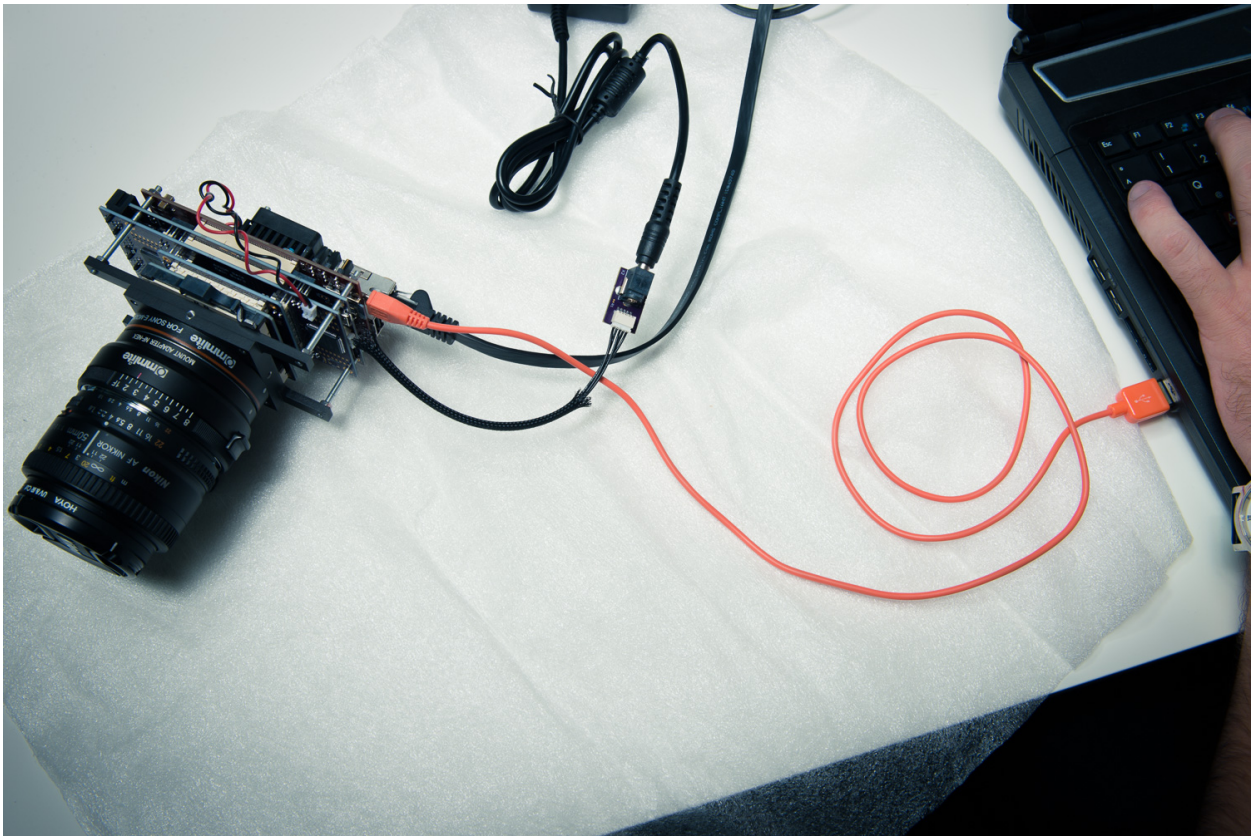
1.1 AXIOM Beta Connector Overview

1.2 Mountpoints

1.3 Accessories and connected devices

2 Operating Basics

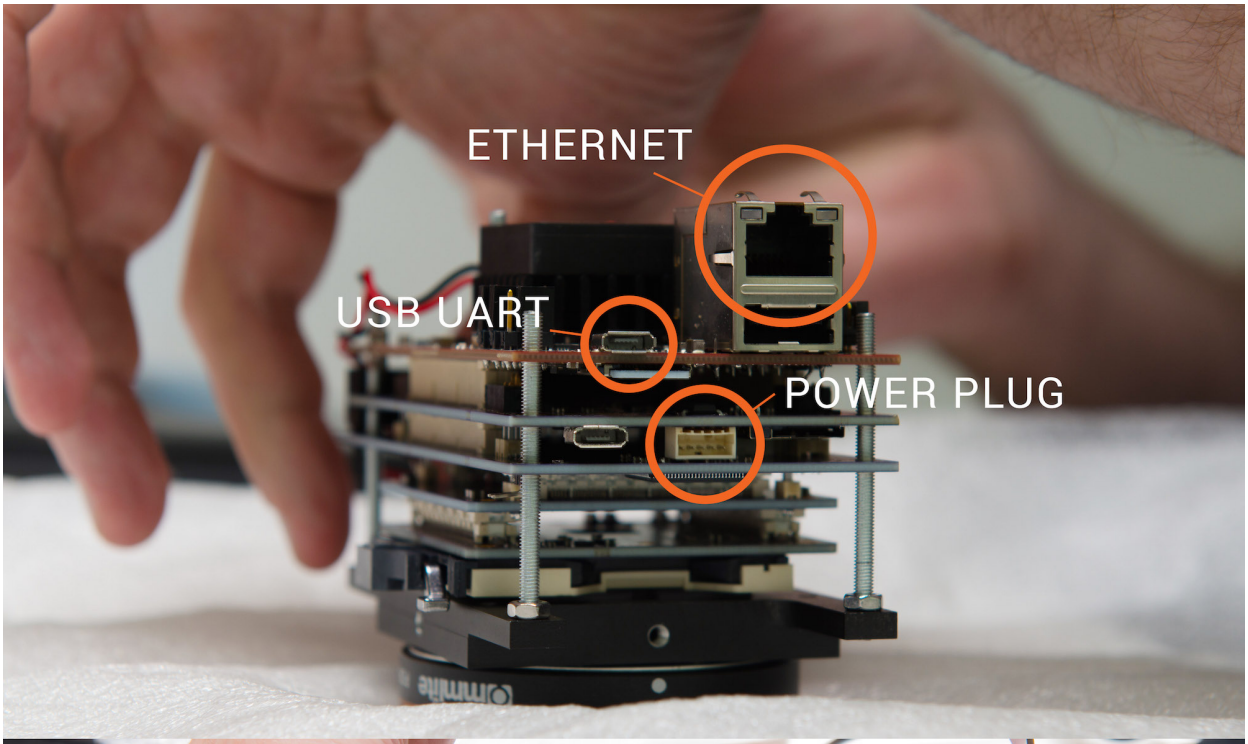
2.1 Prepare your AXIOM Beta for use

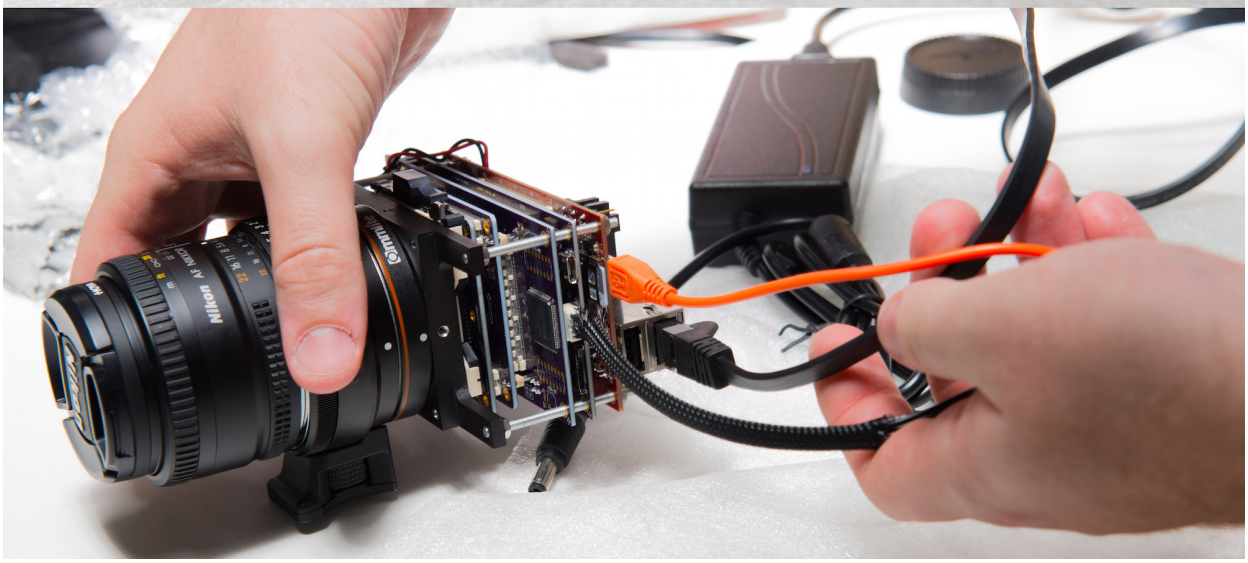


1. Use a micro-USB cable to connect the camera's MicroZed development board (USB UART) to a computer. The MicroZed board is the backmost, red PCB. (There is another micro-USB socket on the Power Board, but that is the JTAG Interface.)

2. Connect the ethernet port on the MicroZed to an ethernet port on your computer. You might have to use an ethernet adapter on newer, smaller machines which come without a native ethernet port.

3. Connect the AC adapter to the camera's Power Board. (The power cord plugs into an adapter that connects to the Power Board; to power the camera off at a later point, you need not disconnect the adapter from the board but can just unplug the cord from the adapter.)





2.2 Prepare your computer for use with AXIOM Beta

Overview - To communicate with your AXIOM Beta camera, you will send it instructions via your computer's command line.

In case you have not worked with a shell (console, terminal) much or ever before, we have prepared detailed instructions to help you get you set up. The steps which need to be taken to prepare your machine sometimes differ between operating systems, so pick the ones that are applicable to you(r system).

Note: Dollar signs placed in front of commands are not meant to be typed in but denote the command line prompt (a signal indicating the computer is ready for user input). It is used in documentation to differentiate

between commands and output resulting from commands. The prompt might look different on your machine (e.g. an angled bracket ¿) and be preceded by your user name, computer name or the name of the directory which you are currently inside.

2.2.1 USB to UART Drivers

For the USB connection to work, you will need drivers for bridging USB to UART (USB to serial). (Under Linux this works out of the box in most distributions) for other operating systems they can be downloaded from e.g. Silicon Labs' website pick the software provided for your OS and install it.

2.2.2 Serial Console

The tool we recommend for connecting to the AXIOM Beta camera via serial port with Mac OS X or Linux is minicom; for connections from Windows machines, we have used Putty.

2.2.2.1 Linux Setup Check if you already have minicom installed on your system by trying to run it:

```
$ minicom
```

Your system will respond with a message like `bash: command not found: minicom` if it's not installed.

Install minicom

Install the minicom package like you'd install other software on your system which could be via a GUI tool or using `aptitude` or `apt-get` (for wich you might need super user rights), e.g.:

```
$ apt-get install minicom
```

or

```
$ sudo apt-get install minicom
```

2.2.2.2 MAC OSX Setup You will want to have Homebrew installed on your system to use `minicom` for serial communication as it is more convenient than using `screen`.

Note: Homebrew is a package manager for Mac, a piece of software that helps you install other software on your Mac machine, particularly software which is readily available on Linux but which does not come in the form of Mac "applications", which you can download via your web browser and simply drop into your Applications folder.

Open Terminal.app (or your preferred terminal emulator if you have another installed). Terminal can be found via e.g. Spotlight search or via the Finder menu: `Go ¿ Utilities ¿ Terminal.app`.

Check if you already have brew installed by entering the brew command:

```
$ brew
```

If you don't have Homebrew installed, your shell will reply with something like `bash: command not found: brew`. Otherwise, it will spit out a list of `brew` commands.

Install Homebrew

To install Homebrew, go to the Homebrew website and follow the install instructions there. You can simply copy the command used for installing Homebrew from their website and paste it into your terminal.

Install minicom

With brew installed, you want to install minicom:

```
$ brew install minicom
```

Homebrew will tell you if you already have minicom installed on your system (e.g. `Warning: minicom-2.7 already installed`), otherwise it will install it for you.

2.2.2.3 Minicom Configuration Once you have minicom installed, you need to configure it in order to talk to the camera.

You can either use the configuration file we prepared or configure it yourself, following our step-by-step instructions.

Use Settings File

Linux

Go to the minicom setup page:

```
$ minicom -s
```

In the "Serial port setup" subpage, check that "Serial Device" 's name is the good one (usually /dev/ttyUSB0 on Linux) and check the baud rate (115200).

Mac

Download the Settings File for Mac, unzip it and place it in the `etc` directory of your minicom install. The minicom installation can be found in the standard directory used by homebrew, `/usr/local/Cellar`, in a subdirectory based on the minicom version number, e.g. `/usr/local/Cellar/minicom/2.7/etc`.

You can also use Homebrew's `info` command to find minicom on your hard disk:

```
$ brew info minicom
```

... which will output general information on the installed package, including its install directory e.g.:

```
minicom: stable 2.7 (bottled)
Menu-driven communications program
https://alioth.debian.org/projects/minicom/
/usr/local/Cellar/minicom/2.7 (17 files, 346.6K) *
  Poured from bottle on ...
```

2.2.3 Serial connection (via USB)

While the AXIOM Beta can be connected to via USB UART (USB to serial), a serial connection is not the preferred way to communicate with the camera but rather in place for monitoring purposes.

However, a serial connection is needed to set up communication via ethernet/LAN (the better suited way to talk to the camera): as the Beta only allows for secure ethernet connections, you will have to connect to the camera via serial port first and copy over your SSH key.

Below are two different methods for connecting to the AXIOM Beta camera, using a program called minicom and an alternative program called screen. We suggest you try them in the order below, so if you can connect with minicom great, if not try screen.

2.2.3.1 Connect using Minicom

Note : You will not be able to use the terminal window you initiate the serial connection in for anything else (it needs to remain open while you access the camera), so it might make sense to open a separate window just for this purpose.

With minicom installed and properly configured, all you need to do is run the following command to start it with the correct settings:

```
$ minicom -8 USB0
```

On successful connection, you will be prompted to enter user credentials (which are needed to log into the camera).

If your terminal remains blank except for the minicom welcome screen/information about your connection settings, try pressing enter. If this still does not result in the prompt for user credentials while testing, we discovered the initial connection with minicom does not always work disconnect the camera from the power adapter, then reconnect it: in your minicom window you should now see the camera's operating system booting up, followed by the login prompt. (From then on, connecting with minicom should work smoothly and at most require you to press enter to make the login prompt appear.)

The default credentials are:

```
user: root
password: beta
```

Alternative tools for serial connection

Before you can use any tool to initiate a serial connection with your Beta camera, you need to know through which special device file it can be accessed.

Once the Beta is connected and powered on (and you installed the necessary drivers), it gets listed as a USB device in the `/dev` directory of your file system, e.g.

`/dev/ttyUSB0` (on Linux)
or
`/dev/cu.SLAB_USBtoUART`

`/dev/tty.SLAB_USBtoUART` (on Mac).
You can use a command such as:

```
$ ls -al /dev | grep -i usb
```

... to list all USB devices currently connected to your machine.

2.2.3.2 Connect using Screen

To connect to the camera, use the command:

```
$ screen file_path 115200
```

... where `file_path` is the full path to the special device file (e.g. `/dev/ttyUSB0` or `/dev/cu.SLAB_USBtoUART`).
You might have to run the command with superuser rights, i.e.:

```
$ sudo screen file_path 115200
```

On successful connection, you will be prompted to enter user credentials needed for logging into the camera. If your terminal remains blank, try pressing enter.

The default credentials are:

```
user: root
password: beta
```

2.2.3.3 Disconnect

To exit the camera's operating system, use:

```
$ exit
```

The result will be a logout message followed by a new login prompt.
To suspend or quit your screen session (and return to your regular terminal window) use one of the following commands:

```
CTRL+a CTRL+z
```

```
CTRL+a CTRL+\
```

2.2.4 Ethernet connection (using SSH)

To access the AXIOM Beta via Ethernet (the preferred way to communicate with it and a method which will also work remotely ie. from a machine which is not directly connected to the camera) authentication via SSH is required.

2.2.4.1 SSH Keys how-to for Linux and Mac

2.2.4.1.1 Storage location/Find existing keys By default, the ssh directory is located at /.ssh, and contains key files called id_rsa and id_rsa.pub, respectively. Check if the directory exists and already contains keys by listing its contents:

```
$ ls -al ~/.ssh
```

If the directory doesn't exist or is empty and you don't have your SSH keys stored elsewhere on your machine, follow the instructions for key creation below.

2.2.4.1.2 SSH key creation Linux and Mac

Linux machines as well as new Macs usually come pre-installed with the tools you need for creating SSH keys. To start the key creation process, use the command:

```
$ ssh-keygen -t rsa -b 4096 -C "yourname@yourmachine"
```

Note: The -C argument is used to add a comment which can help indentify your key as yours/your machine's, which might come in handy once you use other computers to connect to your Beta camera. If you leave it out, your default username/hostname will be used. You can check with:

```
$ echo "$(whoami)@$(hostname)"
```

... either beforehand or in another Terminal window to find out what it is - although you can always change the comment part again later on.

You will be prompted for a file in which to save the keys. To use the default install location (recommended), just press Enter.

Next, you will be asked to enter a passphrase. Using a passphrase means greater security, though you can continue without one by just pressing Enter. In either case you will be asked to confirm your passphrase (by re-entering it if you used one, or pressing Enter again in case you did not).

Subsequently, your keys will be created and saved in the directory you specified (/.ssh by default). You can have your machine print out your public key for you by using the command:

```
$ cat ~/.ssh/id\_rsa.pub
```

The output will look approximately like this:

```
ssh-rsa
    AAAAB3NzaC1yc2EAAAADAQABAAACAQCj8ZHA1ehuCwXvEzyG20Cv0SX1BZ9uyXvON4mDOJJHtG7WolOZG0QPYEPpNxUIFuuvY
    anne@farragut
```

Note: In newer Ubuntu versions (tested with 16.04) it seems that the newly created key is not loaded by the keyserver until you manually run:

```
$ ssh-add
```

2.2.4.2 Get or set IP address

If your network has a DHCP server running somewhere (eg. router) the AXIOM Beta will receive an IP address automatically as soon as it is connected the network.
Otherwise, you will have to set the Beta's IP address manually with the ifconfig command over the serial console (USB).

2.2.4.3 IP address check

While connected to the AXIOM Beta via USB, you can use the command:

```
$ ip a l
```

... to check whether the Beta is currently assigned an IP address. Find the entry which begins with eth0 and check if it contains a line starting with inet followed by an IP address. If it does, this is the IP address the Beta can be reached at.

Example output with no IP address assigned:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0a:35:00:01:26 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::20a:35ff:fe00:126/64 scope link
```

```
10          valid_lft forever preferred_lft forever
```

2.2.4.4 Set IP address If the IP address check is not successful in that it does not produce an IP address you can connect to you will have to set your Beta's IP address manually with the ifconfig command.

While connected to the Beta, you can do that like so:

```
$ ifconfig eth0 192.168.0.9/24 up
```

It does not really matter which IP address you choose as long as it is one allowed for private use (e.g. addresses in the 192.168.x.x range) and you make sure to use the /24 prefix. Note that this will set the IP only until you reboot the camera - then the IP has to be set this same way. A more permanenty solution is using a router and connecting your camera and computer. Then the router assigns IPs using DHCP automatically.

If you now use:

```
$ ip a l
```

... (again) to check for the camera's IP address, you should see the address you assigned listed after inet, e.g.:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0a:35:00:01:26 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.9/24 brd 192.168.0.255 scope global dynamic eth0
10     valid_lft 172739sec preferred_lft 172739sec
    inet6 fe80::20a:35ff:fe00:126/64 scope link
       valid_lft forever preferred_lft forever
```

2.2.4.5 Establish a connection via key Now we have the network configured we need to copy over our SSH key to the Beta.

While connected to the AXIOM Beta via USB, you can use the command:

```
    cd ~/.ssh/
cp authorized\_keys authorized\_keys.orig
```

Strictly speaking the cp (copy) command isn't needed, but it's best practice to always make a copy of a file before editing it - just in case.

The next command will be to add your SSH key to the SSH file which contains the information on who can log into it via SSH without a password.

You will need to first copy (control/command c) - IMPORTANT, make sure that you copy the whole key but do not include the next/new line.

Next, with the command below, you will add your key to the authorized_keys file. The structure of the command below is:

- echo - writes the text after it
- [your key]- pastes (control/command v) your key, so **do not type in [your key]!**
- ¿ - overwrites the existing file and adds just this text
- authorized_keys - make sure to spell this correctly.

As a heads-up, when you paste in your SSH key the terminal window may wrap the text in a way that looks a bit of a messy - this is normal and you can then carry on typing in the rest of the command.

```
echo [yourkey] >> authorized\_keys
```

Alternatively, you can use ssh-copy-id to do it. Don't forget to adapt the IP adress.

```
ssh-copy-id root@192.168.0.9
```

Now you have added you key you can then try and ssh into the Beta.

On your computer use the following command - note the IP address will be the one you set earlier, the one used below is just the example IP.

```
ssh root@192.168.0.9
```

You should now see the following prompt if you have logged on successfully to the Beta.

```
 Last login: Fri Jun 10 15:12:17 2016
sourcing .bashrc ...
[root@beta ~]#
```

Congratulations, you have now set up the Beta's networking and SSH configuration.

The minicom or screen method you used to connect is no longer needed.

2.2.4.6 Password-based authentication If you do not want to use the public/private keypair authentication you can edit /etc/ssh/sshd_config and set:

```
PasswordAuthentication yes
PermitRootLogin yes
```

Note: This has the potential to be a security vulnerability (especially if you do not change the default credentials) and connect your Beta directly to the Internet.

2.2.5 Start the camera

As development of the Beta continues the camera will initialize all systems and train the sensor communication automatically when powered on, but for now you will have to manually start this yourself.

Run the following command:

```
./kick_manual.sh
```

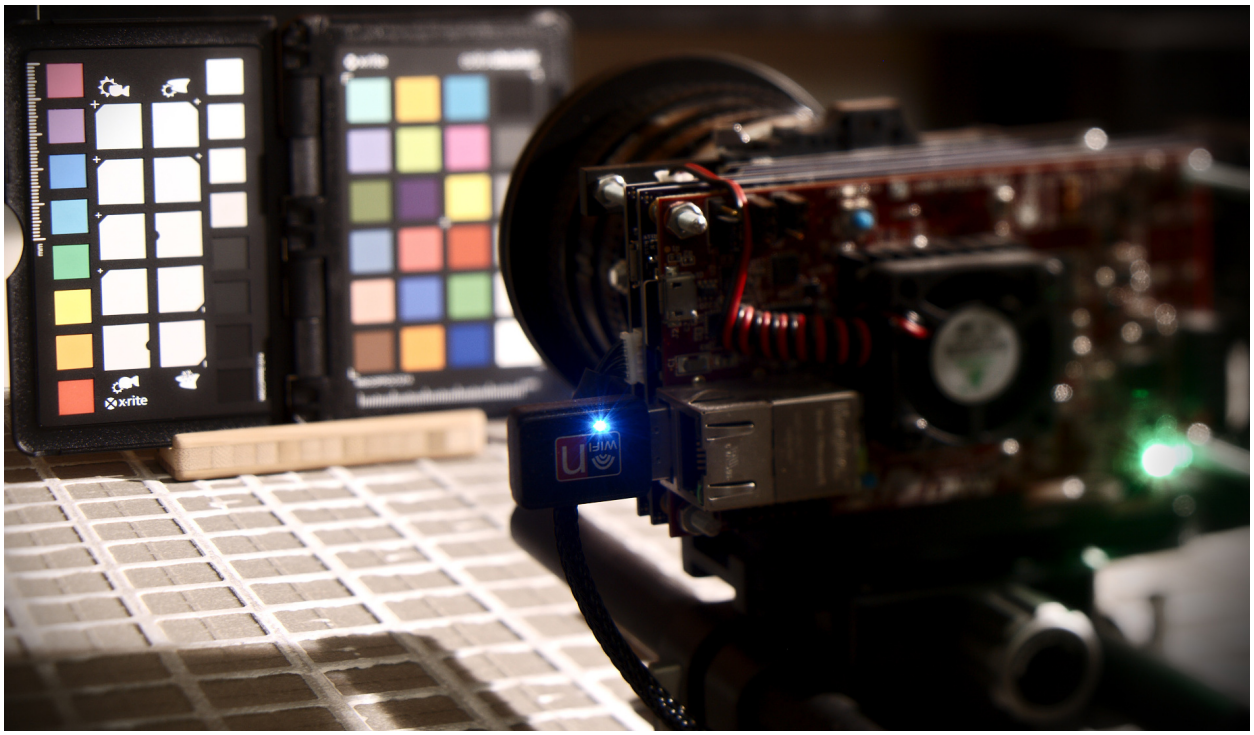
You will see a lot of output, the tail of it is below:

```
..
mapped 0x8030C000+0x00004000 to 0x8030C000.
mapped 0x80000000+0x00400000 to 0xA6961000.
mapped 0x18000000+0x08000000 to 0x18000000.
read buffer = 0x18390200
selecting RFW [bus A] ...
found MXO2-1200HC [00000001001010111010000001000011]
[root@beta ~]#
```

If you look at the back of the camera you will now see a blue LED near the top flashing very fast - this indicates everything is now running.

If you turn the camera off when you turn it back on you will need to re-run this command.

2.2.6 WiFi access point setup



Connecting a USB wifi dongle (that supports Soft-AP) to the AXIOM Beta USB port allows controlling the camera via wireless connection.

In order to do this you must first check your wifi card:

```
ifconfig
```

Turn on wifi card:

```
ifconfig wlan0 up
```

Search wifi essid/hotspot name:

```
iwlist wlan0 scan
```

Setting up Essid and password:

WEP:

```
iwconfig wlan0 essid "yourhotspotname" key yourpassword
```

ASCII:

```
iwconfig wlan0 essid "yourhotspotname" key s:asciikey
```

WPA2 (tested and working) update mirrors database (it is required to connect to the Internet via Ethernet):

```
pacman -Syy
```

Install wpa_supplicant:

```
pacman -S wpa_supplicant
```

Configuration with wpa_passphrase - https://wiki.archlinux.org/index.php/WPA_supplicant create basic configuration file MYSSID = name of your wireless network passphrase = password to connect to your wireless network.

```
wpa_passphrase MYSSID passphrase > /etc/wpa_supplicant/example.conf
```

```
ip link set dev wlan0 up
```

```
wpa_supplicant -B -i wlan0 -c /etc/wpa_supplicant/example.conf
```

If nl80211 driver does not support the given hardware. The deprecated wext driver might still support the device:

```
wpa_supplicant -B -i wlan0 -D wext -c /etc/wpa_supplicant/example.conf
```

```
dhcpcd wlan0
```

Autostart connect https://wiki.archlinux.org/index.php/netctl copy config from example:

```
cp /etc/netctl/example/wireless-wpa /etc/netctl/wireless-wpa
```

Edit the config for you network (interface/ESSID/key):

```
nano /etc/netctl/wireless-wpa
```

First manually check that the profile can be started successfully with:

```
netctl start wireless-wpa
```

If the above command results in a failure, then use:

```
journalctl -xn
netctl status profile
```

... to obtain a more in depth explanation of the failure. (profile might already been started)

Enable:

```
netctl enable wireless-wpa
```

This will create and enable a systemd service that will start when the computer boots. Changes to the profile file will not propagate to the service file automatically. After such changes, it is necessary to reenable the profile:

```
netctl reenable wireless-wpa
```

3 Writing Images

For writing uncompressed full resolution full bitdepth raw images the AXIOM Beta uses a software called cmv_snap3.

It is located in the /root/ directory and writes the images data directly to STDOUT.

cmv_snap3 writes images in the RAW12 format. Writing one image takes a few seconds depending on where the image is written to so this method is not viable for recording video footage other than timelapse.

3.1 Capture Still Images

3.1.1 Parameters

The following parameters are available:

```
./cmv_snap3 -h
This is ./cmv_snap3 V1.10
options are:
-h        print this help message
-8        output 8 bit per pixel
-2        output 12 bit per pixel
-d        dump buffer memory
-b        enable black columns
-p        prime buffer memory
-r        dump sensor registers
-t        enable cmv test pattern
-z        produce no data output
-e <exp>  exposure times
-v <exp>  exposure voltages
-s <num>  shift values by <num>
-S <val>  writer byte strobe
-R <fil>  load sensor registers
```

Examples

Images can be written directly to the cameras internal micro SD card like this (10 milliseconds exposure time, 16bit):

```
./cmv_snap3 -e 10ms > image.raw16
```

Write image plus metadata (sensor configuration) to cameras internal micro SD card (20 milliseconds exposure time, 12bit):

```
./cmv_snap3 -2 -r -e 20ms > image.raw12
```

You can also use cmv_snap3 to change to exposure time (to 5 milliseconds in this example) without actually capturing an image, for the that -z parameter is used to not produce any data output:

```
./cmv_snap3 -z -e 5ms
```

That cmv_snap3 writes data to STDOUT makes it very versatile, we can for example capture images from and to a remote Linux machine connected to the Beta via Ethernet easily (lets assume the AXIOM Betas camera IP is set up as: 192.168.0.9 - SSH access has to be set up for this to work with a keypair)

```
ssh root@192.168.0.9 "./cmv_snap3 -2 -r -e 10ms" > snap.raw12
```

To pipe the data into a file and display it at the same time with imagemagick on a remote machine:

```
ssh root@192.168.0.9 "./cmv_snap3 -2 -r -e 10ms" | tee snap.raw12 | display -size 4096x3072 -
```

Use imagemagick to convert raw12 file into a color preview image:

```
cat test.raw12 | convert \( -size 4096x3072 -depth 12 gray:- \) \( -clone 0 -crop -1-1 \) \( -clo
```

With raw2dng compiled inside the camera you can capture images directly to DNG, without saving the raw12:

```
./cmv_snap3 -2 -b -r -e 10ms | raw2dng snap.DNG
```

Note: Supplying exposure time as parameter is required otherwise cmv_snap3 will not capture an image. The exposure time can be supplied in "s" (seconds), "ms" (milliseconds), "us" (microseconds) and "ns" (nanoseconds). Decimal values also work (eg. "15.5ms").

3.2 Image Overlays

This section covers the mimg version 1.8 (see GitHub repo), not the previous 1.6.
AXIOM Beta features a full HD framebuffer that can be altered from the Linux userspace and is automatically "mixed" with the real time video from the image sensor on HDMI outputs.

The overlay could also be used to draw live histograms/scopes/HUD or menus.

The overlay in more recent firmware revisions supports alpha channel transparency.

3.2.1 Internals

The raw memory image is saved the following way: ¡ch0/12bit¿¡ch1/12bit¿¡ch2/12bit¿¡ch3/12bit¿¡overlay/16bit¿

This means that for every 4 sensels there is only one overlay pixel. Meaning, while the CMV12000 image sensor is 4Kx3K the overlay is Full HD (1080p).

3.2.2 Prepare Images

Convert 24bit PNG (1920x1080 pixels) with transparency to AXIOM Beta raw format:

```
convert input.png rgba:output.rgb
```

Note : This image format is different from RAW12.

3.2.3 mimg

mimg is software running on the camera that's used to load/alter anything related to overlays or test images.

Source code is available on GitHub.

```
This is ./mimg V1.8
options are:
-h        print this help message
-a        load all buffers
-o        load as overlay
-O        load as color overlay
-r        load raw data
-w        use word sized data
-D <val>  image color depth
-W <val>  image width
-H <val>  image height
-P <val>  uniform pixel color
-T <val>  load test pattern
-B <val>  memory mapping base
-S <val>  memory mapping size
-A <val>  memory mapping address
```

Examples:

Clear overlay:

```
./mimg -a -o -P 0
```

Load monochrome overlay:

```
./mimg -o -a file.raw
```

Load color overlay:

```
./mimg -O -a file.raw
```

Enable overlay:

```
gen_reg 11 0x0104F000
```

Disable overlay:

```
gen_reg 11 0x0004F000
```

Old overlays:

Overlays for mimg 1.6 are not compatible anymore. You can use

```
convert -size 1920x1080 -depth 6 rgba:overlay_04.rgb overlay_04.png
```

... to convert them to PNG before continuing with the preparation above.

4 Changing Camera Parameters

Some intro text required

4.1 Setting CMV12000 sensor registers

cmv_reg
Get and Set CMV12000 image sensor registers (CMV12000 sports 128x16 Bit registers).

Details for the sensor datasheet are on GitHub under Datasheets

Examples:

Read register 115 (which contains the analog gain settings):

```
cmv_reg  115
```

Return value:

```
0 x00
```

Means we are currently operating at analog gain x1 = unity gain

Set register 115 to gain x2:

```
cmv_reg 115  1
```

4.2 Setting Exposure Time

To set the exposure time use the cmv_snap3 tool with -z parameter (this will tell the software to not save the image):

```
./cmv_snap3 -e 9.2ms -z
```

Note: The exposure time can be supplied in "s" (seconds), "ms" (milliseconds), "us" (microseconds) and "ns" (nanoseconds). Decimal values also work (eg. "15.5ms").

4.3 Setting gain value

set_gain.sh

Set gain and related settings (ADC range and offsets).
¡¡¡¡¡¡¡ Updated upstream

```
        ./set\_gain.sh 1 ^^J
        ./set\_gain.sh 2 ^^J
        ./set\_gain.sh 3/3 # almost the same as gain 1 ^^J
        ./set\_gain.sh 3 ^^J
        ./set\_gain.sh 4 ^^J
```

=======

```
    ./set_gain.sh 1
    ./set_gain.sh 2
    ./set_gain.sh 3/3 # almost the same as gain 1
    ./set_gain.sh 3
    ./set_gain.sh 4
```

¿¿¿¿¿¿¿ Stashed changes

4.4 Setting Gamma Values

gamma_conf.sh

Set the gamma value:

```
        ./gamma\_conf.sh 0.4
        ./gamma\_conf.sh 0.9
        ./gamma\_conf.sh 1
        ./gamma\_conf.sh 2
```

5 Image metadata

The RAW12 is designed to contain native raw image sensor data for image written by the AXIOM Beta and can optionally also contain an image sensor registers dump (128 x 16bit, big endian) appened at the end of file.
The `-r` command indicates to include the sensor registers when capturing an image (See Section 3.1): ¡¡¡¡¡¡¡ Updated upstream

```
./cmv_snap3 -r -e 10ms > image. raw12
```

=======

```
./cmv_snap3 -r -e 10ms > image. raw12
```

¿¿¿¿¿¿¿ Stashed changes
Show metadata from a RAW12 file (without converting it):

```
raw2dng file.raw12 --dump- regs
```

or, with metadatareader:

```
cat image.raw12 | dd bs=256 skip=73728 | ./ metadatareader
```

Details about the meaning of all image sensor registers can be found in the image sensor datasheet.

5.1 Image Histogram Data

To read and output histogram data from the Betas live image a tool called `cmv_hist3` is available inside the AXIOM Beta.

```
./cmv\_hist3 -h
This is ./cmv_hist3 V1.4
options are:
-h        print this help message
-s        acquire snapshot
-b <num>  number of bins
-d <num>  decimation factor
-r <num>  number of rows
-C <prc>  center sample area
-B <val>  register mapping base
-S <val>  register mapping size
```

The output in 12 bit mode (default) are values in 4096 lines and 4 columns (R, G, B, GB channels)

6 Output

Intro text required

6.1 HDMI

HDMI is a two-way communication protocol and supports many different formats/frequencies/specs. Many monitors/recorders only support a subset of these formats and expect signals to conform to certain values. These values are not documented publicly so we are currently in the process of debugging device compatibility one by one. The good thing is that it's a pure software thing and we can add support and test compatibility with additional devices as time progresses.

In general we discovered that monitors are more flexible when it comes to HDMI (TMDS) freqencies as they just "tune" into (sync to) the provided clock/data rate. Recorders expect signals to be in a much stricter/narrower range and will not work (show "no signal") if there is a minor deviation.
Watch this 33C3 talk by Tim Ansell about Dissecting HDMI to get insight into how HDMI works.

6.1.1 External HDMI Recording

Settings for VSync, HSync, etc. inside the AXIOM Beta can be found in:

```
/root/gen_init. sh
```

For example the Atomos SHOGUN was found to work with these HDMI parameters:

```
scn\_reg  0 2200            # total\_w
scn\_reg  1 1125            # total\_h
scn\_reg  2   60            # total\_f

scn\_reg  4  262            # hdisp\_s
scn\_reg  5 2182            # hdisp\_e
scn\_reg  6   45            # vdisp\_s
scn\_reg  7 1125            # vdisp\_e

scn\_reg  8    0            # hsync\_s
scn\_reg  9 2100            # hsync\_e
scn\_reg 10    4            # vsync\_s
scn\_reg 11    9            # vsync\_e

scn\_reg 32  252            # pream\_s
scn\_reg 33  260            # guard\_s
scn\_reg 34  294            # terc4\_e
scn\_reg 35  296            # guard\_e
```

Currently it is not possible to alter TMDS and Clock frequencies from the userspace (requires new FPGA bitstream).

For the firmware there are two modes available, the 30Hz and 60Hz variant. You can switch between them quite easily.

cmv_hdmi3.bit is the FPGA bitstream loaded for the HDMI interface. We use symlinks to switch this file easily.

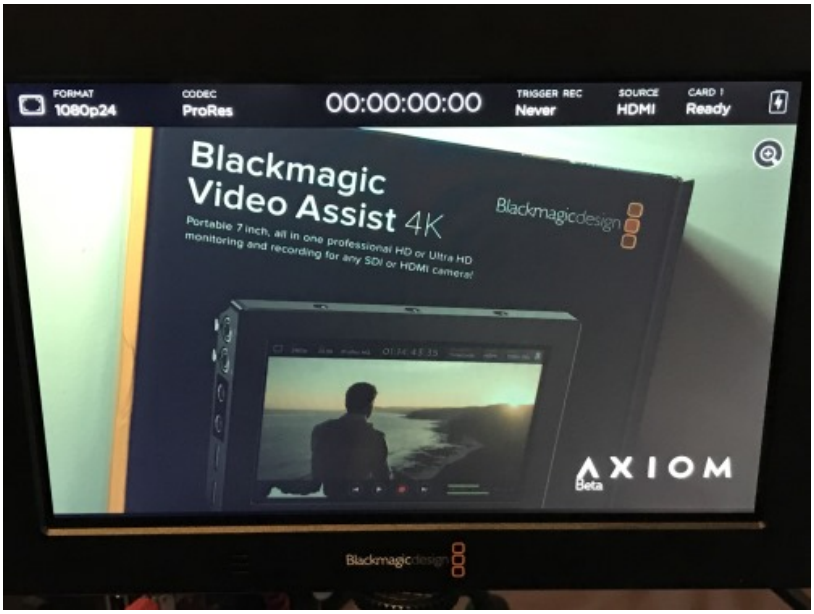
Before doing this, don't forget to check if the files (cmv_hdmi3_60.bit or cmv_hdmi3_30.bit) really exist in the /root folder.

The output is always in 8bpc RGB color space without subsampling (4:4:4). Not all capture devices can manage this.

Other modes like YCrCb, etc. are currently not supported.

6.1.1.1 Devices Confirmed Working Atomos Shogun - AXIOM Beta supports up to 1080p60 Atomos Ninja - AXIOM Beta supports up to 1080p30 Blackmagic Video Assist and Video Assist 4K - AXIOM Beta supports up to 1080p60

Blackmagic Video Assist and Video Assist 4K



Changes for Blackmagic Video Assist and Video Assist 4K, tested on firmware 2.3.1:

edit setup.sh

Add:

```
./gen_init.sh 1080 p60BMVA
```

comment out any other ./gen_init.sh entries.
edit gen_init.sh

```
./gen_init.sh 1080 p60BMVA
```

edit gen_init.sh
Replace:

```
SHOGUN )
```

With:

```
SHOGUN|1080p60BMVA|1080p30BMVA )
```

Add the section below:

```
1080p50BMVA|1080p25BMVA)
scn_reg 0 2640 # total_w
scn_reg 1 1125 # total_h
scn_reg 2 60 # total_f

scn_reg 4 262 # hdisp_s
scn_reg 5 2182 # hdisp_e
scn_reg 6 45 # vdisp_s
scn_reg 7 1125 # vdisp_e

scn_reg 8 0 # hsync_s
scn_reg 9 2100 # hsync_e
scn_reg 10 4 # vsync_s
scn_reg 11 9 # vsync_e

scn_reg 32 252 # pream_s
scn_reg 33 260 # guard_s
scn_reg 34 294 # terc4_e
scn_reg 35 296 # guard_e
;;

1080p24BMVA)
scn_reg 0 2750 # total_w
scn_reg 1 1125 # total_h
scn_reg 2 60 # total_f

scn_reg 4 262 # hdisp_s
scn_reg 5 2182 # hdisp_e
scn_reg 6 45 # vdisp_s
scn_reg 7 1125 # vdisp_e

scn_reg 8 0 # hsync_s
scn_reg 9 2100 # hsync_e
scn_reg 10 4 # vsync_s
scn_reg 11 9 # vsync_e
```

```
    scn_reg 32 252 # pream_s
    scn_reg 33 260 # guard_s
    scn_reg 34 294 # terc4_e
    scn_reg 35 296 # guard_e
    ;;
```

6.1.2 Experimental UHD Raw Recording

Note: This experimental raw mode works only in 1080p60 (A+B Frames) and is only tested with the Atomos Shogun currently.

To measure the required compensations with a different recorder see **Raw processing recorder benchmarking**

This mode requires darkframes which are created in the course of a camera Factory Calibration. Early Betas are not calibrated yet - this step needs to be completed by the user. See **Factory Calibration**.

6.1.2.1 Enable raw recording mode

Input:

```
./hdmi_rectest. sh
```

Inside that script the following command is worth noting:

Enable experimental raw mode:

```
scn_reg 31 0 x0A01
```

Disable experimental raw mode:

```
scn_reg 31 0 x0001
```

If you get an error report like this:

```
Traceback (most recent call last): File "rcn_darkframe.py", line 17, in <module> import
’
```

Make sure the Beta is connected to the Internet via Ethernet and run:

```
pip install  pypng
```

6.1.2.2 Processing Postprocessing software to recover the raw information (DNG sequences) is on github in RAW via HDMI
Required packages: ffmpeg build-essentials
Mac requirements for compiling: gcc4.9(via homebrew):

```
brew install homebrew/versions/ gcc49
```

Also install ffmpeg

To do all the raw processing in one single command (after ffmpeg codec copy processing):

```
./hdmi4k INPUT.MOV - | ./raw2dng --fixrnt --pgm --black=120 frame%05d. dng
```

6.1.3 Experimental UHD Raw Recording

Note: This experimental raw mode works only in 1080p60 (A+B Frames) and is only tested with the Atomos Shogun currently.

To measure the required compensations with a different recorder see **Raw processing recorder benchmarking**

This mode requires darkframes which are created in the course of a camera Factory Calibration. Early Betas are not calibrated yet - this step needs to be completed by the user. See **Factory Calibration**.

6.1.3.1 Enable raw recording mode

Input:

```
./hdmi_rectest. sh
```

Inside that script the following command is worth noting:

Enable experimental raw mode:

```
scn_reg 31 0 x0A01
```

Disable experimental raw mode:

```
scn_reg 31 0 x0001
```

If you get an error report like this:

```
Traceback (most recent call last): File "rcn_darkframe.py", line 17, in <module> import
,
```

Make sure the Beta is connected to the Internet via Ethernet and run:

```
pip install  pypng
```

6.1.3.1.1 Raw processing recorder benchmarking We can analyze footage recorded by the 3rd party recorder but we would need the following:

- Make sure your Beta is running in experimental 4k raw mode (1080p60 with A+B frames)
- Short HDMI captured clip from the 3rd party recorder
- Raw12 still image captured during the HDMI recording

This kind of script is helpful to execute during HDMI recording:

```
# stop HDMI stream:
fil_reg 15 0

# capture image
./cmv_snap3 -r -2 -e 10ms > image.raw12

# start HDMI stream:
fil_reg 15 0 x01000100
```

Taking a snapshot during HDMI recording with the above script will pause the HDMI stream for a few seconds, where it will alternate between two frames. These two frames will be from the same raw data as image.raw12, so they contain all that's needed to figure out what kind of processing the HDMI recorder applies to the image, and how to undo it in order to recover the raw data.

Ideally, the scene should contain fine details (such as tissue, fine print) and rich colors. A color chart (which usually contains some fine print as well) is a very good choice.

... and finally we'd need:

- HDMI captured 1-minute clip with dark frames (lens cap on camera, black cloth covering camera in a dark room)

6.1.3.1.2 Factory Calibration Create a variable containing your Betas IP for easy access:

```
export BETA =192.168.1.101
```

Preperations:

Install on your AXIOM Beta:

```
pacman -S python- numpy
```

Install the following packages on your PC:

```
dcraw  octave
```

For Ubuntu this would look like:

```
sudo apt-get install dcraw  octave
```

Step 1: Check range of the input signal

On the Beta set gain to x1 by running:

```
./set_gain.sh  1
```

Download this Octave file to your PC into your current work directory:

```
wget
    https://raw.githubusercontent.com/apertus-open-source-cinema/misc-tools-utilities/master/darkfram
```

Capture an overexposed image with the Beta and check the levels:

```
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 100ms" > snap.raw12
./raw2dng snap.raw12 --totally-raw
octave
octave:1> a = read\_raw('snap.DNG')
octave:2> prctile(a(:), [0.1 1 50 99 99.9])
```

If everything worked you will get a wall of numbers now.

Lower numbers should be around 50...300 (certainly not zero). Higher numbers should be around 4000, but not 4095.

Repeat for gains 2, 3, 4.

Put this in startup script ie: kick_manual.sh (The systemd service cmv12k is autostarted when boot on and calls the kick.sh and halt.sh scripts on startup and shutdown respectively and those scripts in turn call the kick_manual.sh and halt_manual.sh which also should be used when the cmv12k service is disabled for whatever reason) :

```
./set_gain.sh  1
```

Step 2: RCN calibration

Make sure you have these scripts already in your Beta's /root/ directly.

Clear the old RCN values:

```
ssh root@$BETA "./rcn_clear.py"
```

Now you need to make sure that your Beta is not capturing any light (ideally not a single photon should hit the sensor) :

1. lose the lens aperture as far as possible
2. Attach lens cap
3. Put black lens bag over Beta
4. Turn off all lights in the room - do this at night or in a completely dark room

Take 64 dark frames at 10ms, gain x1.

```
./set\_gain.sh 1
fil_reg 15 0 # disable HDMI stream
for i in `seq 1 64`; do
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 10ms" > dark-x1-10ms-$i.raw12
done
fil_reg 15 0x01000100  # enable HDMI stream
```

Compute a temporary dark frame for RCN calibration:

```
raw2dng --swap-lines --no-blackcol --calc-darkframe dark-x1-10ms-*. raw12
```

This should process quite quickly and output something like the following at the end:

```
Averaged 64 frames exposed from 12.00 to 12.00 ms. Could not compute dark current. Plea
.
```

Rename and upload darkframe to your Beta:

```
mv darkframe-x1.pgm darkframe-rcn.pgm
scp darkframe-rcn.pgm root@$BETA:/root/
```

Set the RCN values:

```
ssh root@$BETA "./rcn\_darkframe.py darkframe-rcn.pgm"
```

Put this in startup script ie : kick_manual.sh :

```
./rcn\_darkframe.py darkframe-rcn.pgm
```

If you get an error report like this:

```
Traceback (most recent call last):
File "rcn\_darkframe.py", line 17, in <module>
```

```
import png
ImportError: No module named 'png'
```

Make sure the Beta is connected to the Internet via Ethernet and run:

```
pip install  pypng
```

and then run the python script again.

Validation

Method 1:

Put a lens cap on the camera and check the image on a HDMI monitor.

In the camera set the matrix gains to:

```
./mat4_conf.sh 20 0 0 0 0 10 10 0 0 10 10 0 0 0 0 10 0 0 0  0
```

run:

```
./rcn_clear. py
```

The static noise profile should be visible.

run:

```
./rcn_darkframe.py darkframe-rcn. pgm
```

The static noise profile should be gone. You will still see dynamic row noise (horizontal lines flickering) - thats expected.

Method 2:

This method is now entirely automated with running one script inside the camera: https://github.com/apertus-open-source-cinema/beta-software/blob/master/beta-scripts/rcn_validation.sh

Capture one darkframe without compensations:

```
ssh root@$BETA "./rcn\_clear.py"
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 10ms" > dark-check-1.raw12
```

Capture one darkframe with compensations:

```
ssh root@$BETA "./rcn\_darkframe.py darkframe-rcn.pgm"
```

```
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 10ms" > dark-check-2.raw12
```

Then use raw2dng to analyze the differences:

```
raw2dng --no-darkframe --check-darkframe dark-check-1.raw12
raw2dng --no-darkframe --check-darkframe dark-check-2.raw12
```

With the compensated snapshot the column noise should disappear, and only row noise left should be dynamic (not static). Visual inspection: the dark frame should have only horizontal lines, not vertical ones.

Sample output:

```
Average     : 127.36              # about 128, OK
Pixel noise : 5.44                # this one is a bit high because we only corrected
    row and column offsets (it's OK)
Row noise   : 2.30 (42.2%)        # this one should be only dynamic row noise - see
    Method 3 below.
Col noise   : 0.20 (3.8%)         # this one is very small, that's what we need to
    check here}
```

Method 3:

Capture 2 frames:

```
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 10ms" > dark-check-1.raw12
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 10ms" > dark-check-2.raw12
```

Convert the two darkframes with raw2dng:

```
raw2dng dark-check -*
```

Make sure you have the required octave function file in place:

```
wget
    https://raw.githubusercontent.com/apertus-open-source-cinema/misc-tools-utilities/master/darkfram
```

Also you need to install the octave "signal" and "control" packages from http://octave.sourceforge.net/packages.php then, inside octave, run to install:

```
pkg install  package_name
```

To check whether the entire row noise is dynamic, load the two raw images in octave and check the autocorrelation between the two row noise samples:

```
pkg load signal
```

```
a = read\_raw('dark-check-1.DNG');
b = read\_raw('dark-check-2.DNG');
ra = mean(a'); ra = ra - mean(ra);
rb = mean(b'); rb = rb - mean(rb);
xcov(ra, rb, 0, 'coeff')
```

Result should be very small (about 0.1 or lower). When running this check on two uncalibrated dark frames, you will get around 0.8 - 0.9.

Step 3: Dark frame calibration

Make sure the RCN calibration from previous steps is in place before continueing here.

Take dark frames at various exposure times and gains.

```
for i in 1 2 3 4; do
for e in `seq 1 100`; do
for g in 1 2 3 4; do
ssh root@$BETA "./set\_gain.sh $g"
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e ${e}ms" > dark-x${g}-${e}ms-$i.raw12
done
done
done
```

Compute dark frames for each gain:

```
raw2dng --swap-lines --calc-dcnuframe dark-x1-*.raw12 raw2dng --swap-lines --calc-dcnuf
raw12
```

Save the following files (N=1..4):

```
darkframe-xN.pgm dcnuframe-xN. pgm
```

These files should be used in postprocessing. Place them in the directory where you capture raw12 files, so raw2dng will use them.

Validation

On the same dark frames, or - even better - on a new set of dark frames, run:

```
raw2dng --swap-lines --check-darkframe dark*.raw12 > dark-check. log
```

Upload the log for detailed analysis.

Typical good values are:

average value: close to 128

pixel noise: about 3 or 4 (may increase at longer exposure times)

row noise and column noise: similar to Step 2

Step 4: Color profiling

Set gain x1:

```
ssh root@$BETA "./set_gain.sh 1"
```

Take a picture of the IT8 chart, correctly exposed.

Edit the coordinates and the raw file name in calib_argyll.sh (https://github.com/apertus-open-source-cinema/misc-tools-utilities/blob/master/color-calibration/calib_argyll.sh).

```
ssh root@$BETA "./cmv\_snap3 -2 -b -r -e 10ms" > it8chart.raw12
./calib\_argyll.sh IT8
```

Save the following files:

- ICC profile (*.icc)
- OCIO configuration (copy/paste from terminal) + LUT file (*.spi1d)

Validation

Render the IT8 chart in Blender, using the OCIO configuration.

Same with the ICC profile (Adobe? RawTherapee? What apps support ICC?)

(todo: detailed steps)

Step 5: HDMI dark frames

Record a 1-minute clip with lens cap on.

Average odd and even frames.

(todo: polish and upload the averaging script)

(todo: check if the HDMI dark frames can be computed from regular dark frames)

Results: darkframe-hdmi-A.ppm and darkframe-hdmi-B.ppm.

Step 6: HDMI filters for raw recovery

This calibration is for the recorder, not for the camera. It's for recovering the original raw data from the HDMI, so it has nothing to do with sensor profiling and such.

Record some scene with high detail AND rich colors.

Take a raw12 snapshot in the middle of recording. The HDMI stream will pause for a few seconds.

Upload two frames from the paused clip, together with the raw12 file. This calibration will be hardcoded in hdmi4k.

The two frames must be in the native format of your video recorder (not DNG). You should be able to cut the video with ffmpeg -vcodec copy.

6.1.4 EDL Parser

This script can take EDLs to reduce the raw conversion/processing to the essential frames that are actually used in an edit. This way a finished video edit can be converted to raw DNG sequences easily.

Requirements: ruby

```
puts "BEFORE EXECUTION, PLS FILL IN YOUR WORK DIRECTORY IN THE SCRIPT
    (path\_to\_workdir)"


puts "#!/bin/bash"
i=0
ffmpeg\_cmd1 = "ffmpeg -i "

tc\_in = Array.new
tc\_out = Array.new
clip = Array.new

file = ARGV.first
ff = File.open(file, "r")

ff.each\_line do |line|
 clip << line.scan(/NAME:\s(.+)/)
 tc\_in <<
     line.scan(/(\d\d:\d\d:\d\d:\d\d).\d\d:\d\d:\d\d:\d\d.\d\d:\d\d:\d\d:\d\d.\d\d:
 \d\d:\d\d:\d\d/)tc\_out <<
     line.scan(/\s\s\s\d\d:\d\d:\d\d:\d\d\s(\d\d:\d\d:\d\d:\d\d)/)

end
c=0
clip.delete\_at(0)
clip.each do |fuck|
 if clip[c].empty?
      tc\_in[c] = []
      tc\_out[c] = []
 end
 c=c+1
end
```

```
30
    total\_frames = 0
    t\c_in = tc_in.reject(&:empty?)
    tc\_out = tc_out.reject(&:empty?)
    clip = clip.reject(&:empty?)
35  tc\_in.each do |f|
    tt\_in = String.new
    tt\_out = String.new
     tt\_in = tc_in[i].to\_s.scan(/(\d\d)\D(\d\d)\D(\d\d)\D(\d\d)/)
     tt\_out = tc_out[i].to\_s.scan(/(\d\d)\D(\d\d)\D(\d\d)\D(\d\d)/)
40   framecount =
     ((tt\_out[0][0].to\_i-tt\_in[0][0].to_i)*60*60*60+(tt\_out[0][1].to\_i-tt\_in[0]
     [1].to\_i)*60*60+(tt\_out[0][2].to\_i-tt_in[0][2].to\_i)*60+(tt\_out[0][3].to\_i
     -tt\_in[0][3].to\_i))
     framecount = framecount + 20
45   tt\_in\_ff = (tt\_in[0][3].to\_i*1000/60)
     frames\_in =
         tt\_in[0][0].to\_i*60*60*60+tt\_in[0][1].to\_i*60*60+tt\_in[0][2].to\_i*60+tt
     \_in[0][3].to\_i
     frames\_in = frames\_in - 10
     new\_tt\_in = Array.new
50   new\_tt\_in[0] = frames\_in/60/60/60
     frames\_in = frames\_in - new\_tt\_in[0]*60*60*60
     new\_tt\_in[1] = frames\_in/60/60
     frames\_in = frames\_in - new\_tt\_in[1]*60*60
     new\_tt\_in[2] = frames\_in/60
55   frames\_in = frames\_in - new\_tt\_in[2]*60
     new\_tt\_in[3] = frames\_in
     frames\_left =
         (tt\_in[0][0].to\_i*60*60*60+(tt\_in[0][1].to\_i)*60*60+(tt\_in[0][2].to\_i)
     *60+(tt\_in[0][3].to\_i))-10
     new\_frames = Array.new
60   new\_frames[0] = frames\_left/60/60/60
     frames\_left = frames\_left - new\_frames[0]*60*60*60
     new\_frames[1] = frames\_left/60/60
     frames\_left = frames\_left - new\_frames[1]*60*60
     new\_frames[2] = frames\_left/60
65   frames\_left = frames\_left - new\_frames[2]*60
     new\_frames[3] = frames\_left
     tt\_in\_ff\_new = (new\_frames[3]*1000/60)

     clip[i][0][0] = clip[i][0][0].chomp("\r")
70   path\_to\_workdir = "'/Volumes/getztron2/April Fool 2016/V'"
     mkdir = "mkdir #{i}\n"
     puts mkdir
     ff\_cmd\_new = "ffmpeg -ss #{sprintf '%02d', new\_frames[0]}:#{sprintf '%02d',
         new\_frames
     [1]}:#{sprintf '%02d', new\_frames[2]}.#{sprintf '%02d', tt\_in\_ff\_new} -i
         #{path\_to\_
75   workdir}/#{clip[i][0][0].to\_s} -frames:v #{framecount} -c:v copy p.MOV -y"
     puts ff\_cmd_new
     puts "./render.sh p.MOV&&\n"
     puts "mv frame*.DNG #{i}/"
```

```
    hdmi4k\_cmd = "hdmi4k #{path\_to\_workdir}/frame*[0-9].ppm --ufraw-gamma
        --soft-film=1.5 --fixrnt --offset=500&&\n"
80
    ff\_cmd2 = "ffmpeg -i #{path\_to\_workdir}/frame%04d-out.ppm -vcodec prores
        -profile:v 3 #{clip[i][0][0]}\_#{i}\_new.mov -y&&\n"
    puts "\n\n\n"
    i=i+1
    total\_frames = total\_frames + framecount
85  end

  puts "#Total frame: count: #{total\_frames}"
```

Pipe it to a Bash file to have a shell script.

Note from the programmer: This is really unsophisticated and messy. Feel free to alter and share improvements.

6.1.5 cmv perf3

cmv perf 3 text required

6.2 SDI

SDI Instructions required - ETA Dec 2017.

6.3 Modes

Note: Modes like YCrCb, etc. are currently not supported.

6.3.1 1080p60/1080p50 Mode

Enable:

```
rm -f cmv_hdmi3.bit ln -s cmv_hdmi3_60.bit cmv_hdmi3.bit sync reboot  now
```

6.3.2 1080p30/1080p25 Mode

Enable:

```
rm -f cmv_hdmi3.bit ln -s cmv_hdmi3_30.bit cmv_hdmi3.bit sync reboot  now
```

6.4 Generator and HDMI Output

Independet of the firmware you can switch the rate of the generator. In setup.sh you can change the generator resolution and framerate.

After changing the generator mode, make sure to restart it:

```
./halt_manual.sh && ./kick_manual. sh
```

```
./gen_init.sh 1080p60 ./gen_init.sh 1080p50 ./gen_init.sh 1080 p25
```

To enable the shogun mode, which is only possibly by current hardware:

```
./gen_init.sh  SHOGUN
```

1080p25 mode is known to work on the Shogun if using the SHOGUN profile and then setting:

```
scn_reg 0  2640
```

In Shogun mode, the exposure (shutter) is synced to the output frame rate, but can be a multiple, i.e. with 60FPS output, it can be 60, 30, 20, 15, 12, ... The exposure time (shutter angle if divided by FPS) is entirely controlled by the sensor at the moment.

Note that the firmware controls the shutter, not the generator.

In the future, this will be combined and processed by only one piece of software.

6.5 Stopping and Starting HDMI Live-stream

Stop HDMI live stream:

```
fil_reg 15  0
```

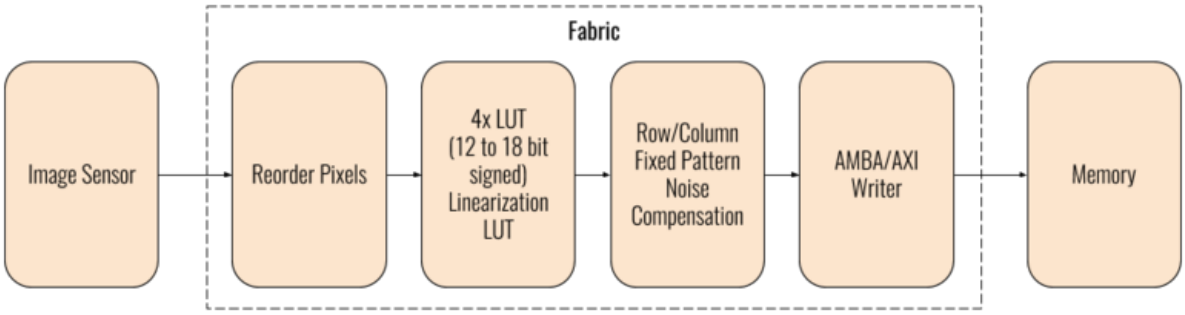
Start HDMI live stream:

```
fil_reg 15 0 x01000100
```

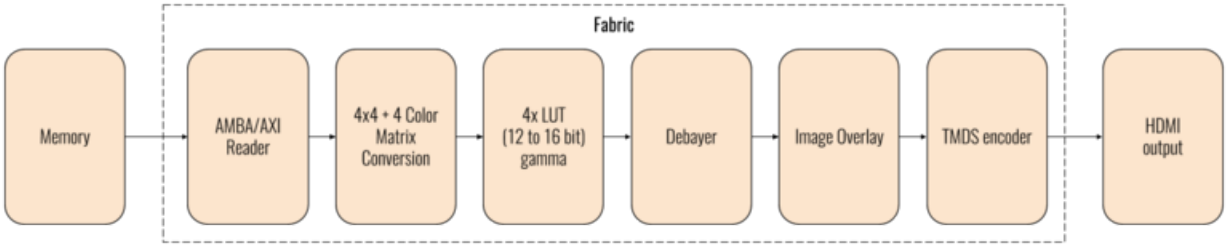
7 Processing

Overview text required.

7.1 Image Acquisition Pipeline



7.2 HDMI Image Processing/Output Pipeline



7.3 SDI Image Processing/Output Pipeline

ToDo - ETA Dec 2017.

7.4 Image Processing Nodes

Debayering

A planned feature is to generate this FPGA code block with "dynamic reconfiguration" meaning that the actual debayering algorithm can be replaced at any time by loading a new FPGA binary block at run-time. This tries to simplify creating custom debayering algorithms with a script like programming language that can be translated to FPGA code and loaded into the FPGA dynamically for testing.

Peaking

Peaking marks high image frequency areas with colored dot overlays. These marked areas are typically the ones "in-focus" currently so this is a handy tool to see where the focus lies with screens that have lower resolution than the camera is capturing.

Handy Custom Parameters:

- color
- frequency threshold

Potential Problems:

There are sharper and softer lenses so the threshold depends on the glass currently used. For a sharp lens the peaking could show areas as "in-focus" if they actually aren't and for softer lenses the peaking might never show up at all because the threshold is never reached.

8 Converting

Overview text required.

8.1 RAW12 to PGM

Converts a video file recorded in AXIOM raw to a PGM image sequence and applies the darkframe (needs to be created beforehand).

Currently clips must go through ffmpeg before hdmi4k can read them:

```
ffmpeg -i CLIP.MOV -c:v copy OUTPUT. MOV
```

To cut out a video between IN and OUT with ffmpeg but maintaing the original encoding data:

```
ffmpeg -i CLIP.MOV -ss IN_SECONDS -t DURATION_SECONDS -c:v copy OUTPUT. MOV
```

```
 hdmi4k
HDMI RAW converter for Axiom BETA

Usage:
  ./hdmi4k clip.mov
  raw2dng frame*.pgm [options]

Calibration files:
  hdmi-darkframe-A.ppm, hdmi-darkframe-B.ppm:
  averaged dark frames from the HDMI recorder (even/odd frames)

Options:
-                   : Output PGM to stdout (can be piped to raw2dng)
--3x3               : Use 3x3 filters to recover detail (default 5x5)
--skip              : Toggle skipping one frame (try if A/B autodetection fails)
--swap              : Swap A and B frames inside a frame pair (encoding bug?)
--onlyA             : Use data from A frames only (for bad takes)
--onlyB             : Use data from B frames only (for bad takes)
```

8.2 RAW12 to DNG

Converts AXIOM Beta raw image to DNG.

```
Inhalt der Datei "Datei.Endung"
23142511
dasas
dasdsa
21512fasf32
rw325tgs
```

Example:

```
./raw2dng --fixrnt --pgm --black=120 frame%05d. dng
```

Compiling raw2dng
Compiling raw2dng on a 64bit system requires the gcc-multilib package.
Ubuntu:

```
sudo apt-get install gcc- multilib
```

openSUSE:

```
sudo zypper install gcc-32bit libgomp1-32 bit
```

AXIOM Beta:
1. Acquire the source from https://github.com/apertus-open-source-cinema/misc-tools-utilities/tree/master/raw2dng.
2. Copy files to AXIOM Beta.
3. Remove `-m32` from Makefile.
4. Run `make` inside camera.

9 Maintenance

Overview text required.

9.1 Firmware

The entire AXIOM Beta firmware is stored on a Micro SD card. This means that the camera's operating system can be swapped out easily and that external recovery of the camera entire software, in case anything went wrong eg. during flashing, is always possible.

It also means that experimental features can easily be tried and tested simply by popping in a different card into the camera.

9.1.1 Firmware Backup

The entire camera firmware is stored on a Micro SD card plugged into the Microzed. To back-up the entire firmware we plug in the Micro SD card into a Linux PC and do the following:
1. Find out which device the micro SD card is:

```
cat /proc/partitions  mount
```

... should give you a list of all connected devices. Lets assume in our case that the card is /dev/sdc.

2. Make sure the card is unmounted (all 3 partitions):

```
umount /dev/sdc1 umount /dev/sdc2 umount /dev/ sdc3t
```

3. clone the entire card to a file:

```
ddrescue /dev/sdc sdimage.img sdimage. log
```

Here is a guide that covers doing the same on Mac and Windows: http://raspberrypi.stackexchange.com/questions/311/how-do-i-backup-my-raspberry-pi

9.1.2 Firmware Restore

Again you need to know the device path of your sd card, then (assuming in our case its /dev/sdb) run:

```
sudo dd if=sdimage.img of=/dev/sdb bs=4 M
```

9.2 Image Sensor cleaning

We are using the green sensor cleaning swabs:

It comes with two cleaning solutions: one for dust and one for any oil based residues (finger prints, etc.)

On this page we try to collect typical sensor contamination images, guides how to spot and get rid of them.

The best lighting conditions to spot contamination seems to be mid grey, you can take off or slightly turn the lens to make sure the contamination is not on any lens glass element.

Vertical streaks:

These are the result of using the dust cleaning solution on the sensor. Likely the streaks are oil based contaminations. Use the red bottle "smear away" and a swap to clean the sensor.

10 Installations

Overview text required.

10.1 Installing a webserver

Installing required packages

Make sure the AXIOM Beta is connected to the internet and then on the commandline:

Update mirrors database:

```
pacman - Syy
```

Install webserver:

```
pacman -S lighttpd php php- cgi
```

Start the webservice:

```
systemctl start  lighttpd
```

Write any pending changes to the file system:

```
sync
```

10.2 Configuring a webserver

This follows the guide from the lighttpd archlinux wiki page: https://wiki.archlinux.org/index.php/lighttpd

```
mkdir /etc/lighttpd/conf.d/ nano /etc/lighttpd/conf.d/cgi. conf
```

... and place the following content in the file:

```
server.modules += ( "mod_cgi" )

cgi.assign = (".pl" => "/usr/bin/perl",
".cgi" => "/usr/bin/perl",
".rb" => "/usr/bin/ruby",
".erb" => "/usr/bin/eruby",
".py" => "/usr/bin/python",
".php" => "/usr/bin/php-cgi")
```

```
index-file.names += ("index.pl", "default.pl",
"index.rb", "default.rb",
"index.erb", "default.erb",
"index.py", "default.py",
"index.php", "default.php")
```

For PHP scripts you will need to make sure the following is set in /etc/php/php.ini

```
cgi.fix_pathinfo =  1
```

In your Lighttpd configuration file, /etc/lighttpd/lighttpd.conf add:

```
include "conf.d/cgi.conf"
```

Create a new configuration file /etc/lighttpd/conf.d/fastcgi.conf

```
# Make sure to install php and php-cgi. See:
# https://wiki.archlinux.org/index.php/Fastcgi_and_lighttpd#PHP

server.modules += ("mod_fastcgi")

# FCGI server
# ===========
#
# Configure a FastCGI server which handles PHP requests.
#
index-file.names += ("index.php")
fastcgi.server = (
# Load-balance requests for this path...
".php" => (
# ... among the following FastCGI servers. The string naming each
# server is just a label used in the logs to identify the server.
"localhost" => (
"bin-path" => "/usr/bin/php-cgi",
"socket" => "/tmp/php-fastcgi.sock",
# breaks SCRIPT_FILENAME in a way that PHP can extract PATH_INFO
# from it
"broken-scriptfilename" => "enable",
# Launch (max-procs + (max-procs * PHP_FCGI_CHILDREN)) procs, where
# max-procs are "watchers" and the rest are "workers". See:
#
https://redmine.lighttpd.net/projects/1/wiki/frequentlyaskedquestions#How-many-php-CGI-pr
"max-procs" => 4, # default value
"bin-environment" => (
"PHP_FCGI_CHILDREN" => "1" # default value
```

```
30  )
    )
    )
    )
```

Make lighttpd use the new configuration file /etc/lighttpd/lighttpd.conf

```
include "conf.d/fastcgi.conf"
```

Restart lighttpd:

```
systemctl restart  lighttpd
```

To test php create a file: /src/http/index.php with content:

```
<?php phpinfo();  ?>
```

... and open this IP address of your AXIOM Beta in a browser. If you see the php info status page everything worked successfully.

10.3 Installing AXIOM Beta Web GUI software

Download this repository - https://github.com/apertus-open-source-cinema/beta-software

1. Copy all files from the http directory of the repository to your AXIOM Beta /srv/http/ directory.
2. Copy all files from the beta-scripts directory of the repository to your AXIOM Beta /root/ directory.

Edit /etc/sudoers files:

Under the line:

```
root ALL=(ALL)  ALL
```

Add:

```
http ALL=(ALL) NOPASSWD:  ALL
```

This allows the http user to do anything with the system so it can be considered a security vulnerability - but for development this should not be an issue, later on we will define the http priviledges more securely.

For testing sudoers:

```
sudo -u http sudo  whoami
```

If it returns `root` then you are all set.

This should provide you with a working webbased GUI.

Note : `lighttpd` does not start automatically when the AXIOM Beta boots, this still needs to be configured:

```
systemctl enable  lighttpd
```

Note also: Opening any websites that read image sensor registers before initializing the image sensor `kick_manual.sh` will freeze/crash the camera.

10.4 Packet Manager Pacman

Update all package definitions and the database from the Internet:

```
pacman - Sy
```

Important: Careful with upgrading existing packages. For example the Kernel used in the AXIOM Beta is custom developed - if you upgrade Arch Linux to the latest off the shelf Kernel you will BRICK your camera firmware.

Install lighttp webserver on the Beta:

```
pacman -S  lighttpd
```

Install PHP on the Beta:

```
pacman -S php php- cgi
```

Follow these instructions: https://wiki.archlinux.org/index.php/lighttpd#PHP
Start the webserver:

```
systemctl start  lighttpd
```

11 Colour Science

Overview text required.

11.1 Black Calibration

Black Calibration is a term that describes finding the sensor output value, per pixel, in the absence of any illumination.
It covers:

- Dark frame subtraction
- Dark current compensation
- Using black reference columns (called "optical black" by other manufacturers) to find the black level and fine-tune static offsets.

Note: Black reference columns can be used to reduce row noise as well. See Section 11.2 Pattern Noise.

11.1.1 Calibration methods

11.1.1.1 Dark Frame Subtraction This is a basic technique: take a picture with the lens cap on, and subtract it from your image. To make really sure no light is reaching the sensor, also cover the entire camera with something.

How many dark frames?

Problem: if you take only one dark frame, it will also contain read noise (assummed to be Gaussian and uncorrelated with the read noise from other frames). Therefore, subtracting only one image will actually increase the noise in the final output, by sqrt(2).

Solution: use a master dark frame, averaged from many images.

How many?

If you take N images, the signal will be multiplied by N, and the noise will be multiplied by sqrt(N). Therefore, the SNR will increase by log2(sqrt(N)) stops.

So, 16 dark frames will reduce the read noise in the dark frame by 2 stops, 64 frames by 3 stops, and 256 frames by 4 stops.

Okay, but how much noise will be added to the output image?

Let's say the read noise stdev in one image is r, so a dark frame averaged from N frames will have noise

```
stdev = r/sqrt(N )
```

. Therefore, the noise in the output image will be:

```
sqrt(r^2 + r^2/N) = r * sqrt(1 + 1/N )
```

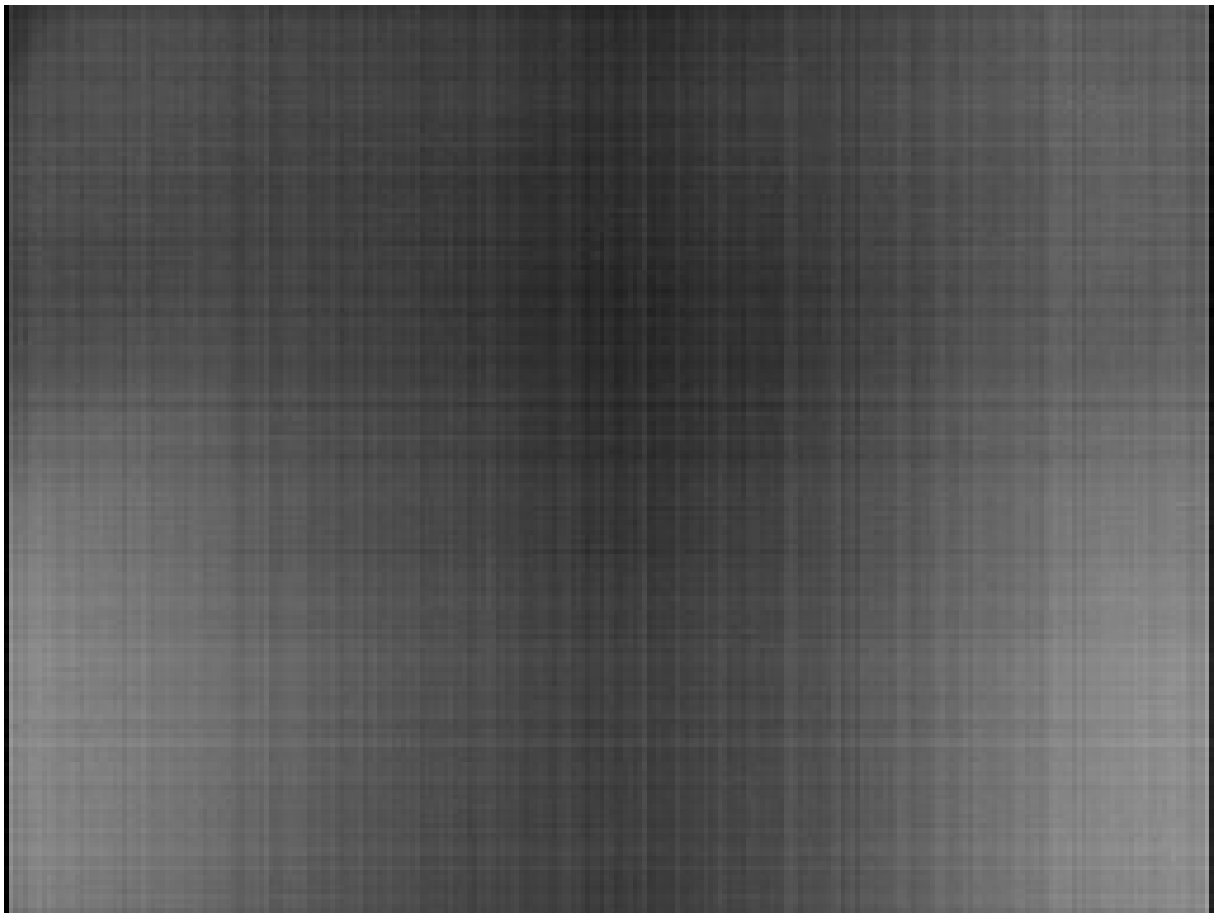
.

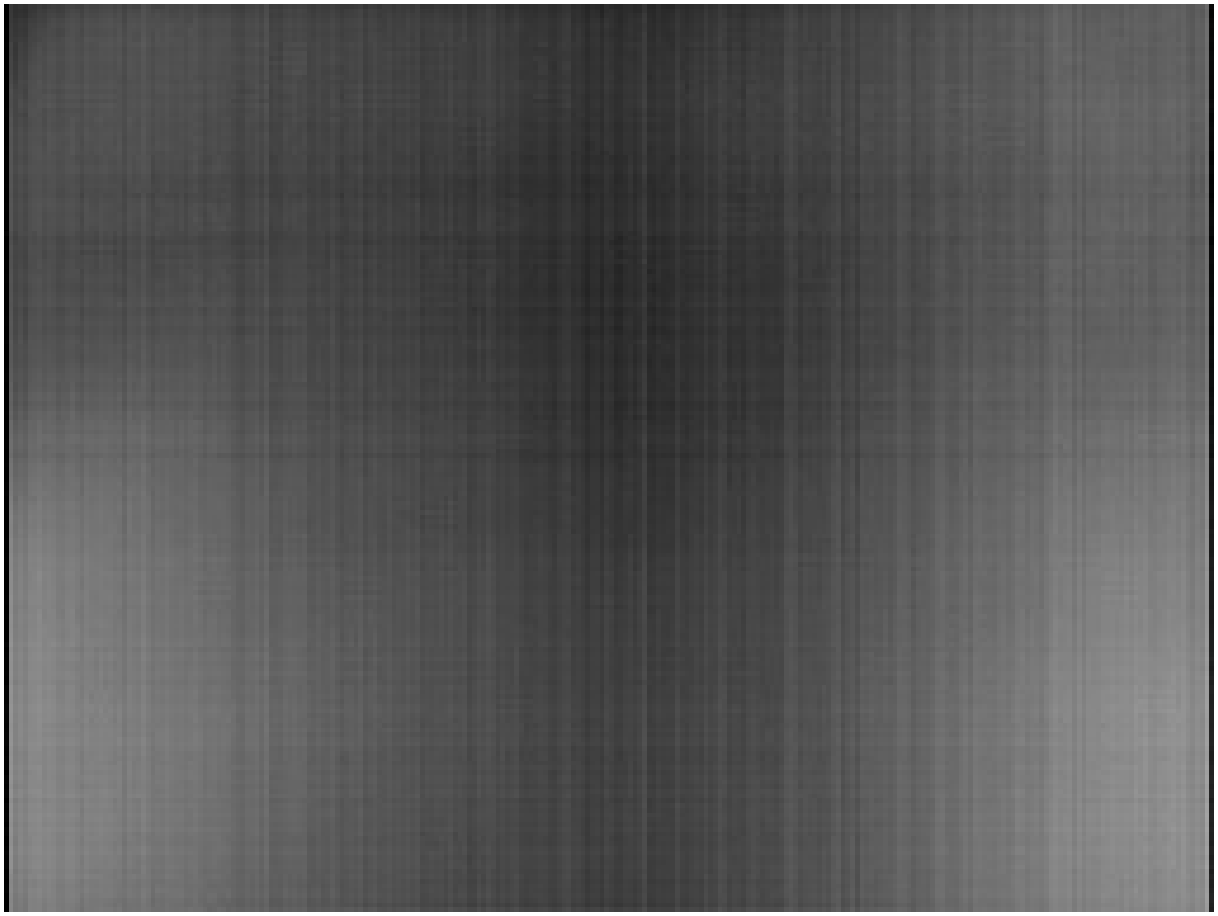
Check it in octave:

```
octave:1>  N = 4;
octave:2>  a = randn(1,1000000);     # one Gaussian noise sample, with mean=0 and stdev=1
octave:3>  b = randn(N,1000000);     # N noise samples
octave:4>  std(a + mean(b))          # add one noise sample to N averaged noise samples
ans =  1.1174
octave:5>  sqrt(1 + 1/4)             # compare with the theoretical result
ans =  1.1180
```

So, it seems that averaging a small number of dark frames will not introduce significant noise in your images (4 should be enough if you are in a hurry, and 16 should give a very good result).

Example: one dark frame at 6ms x1 (same image as above), vs 4 darkframes at 6ms, averaged.





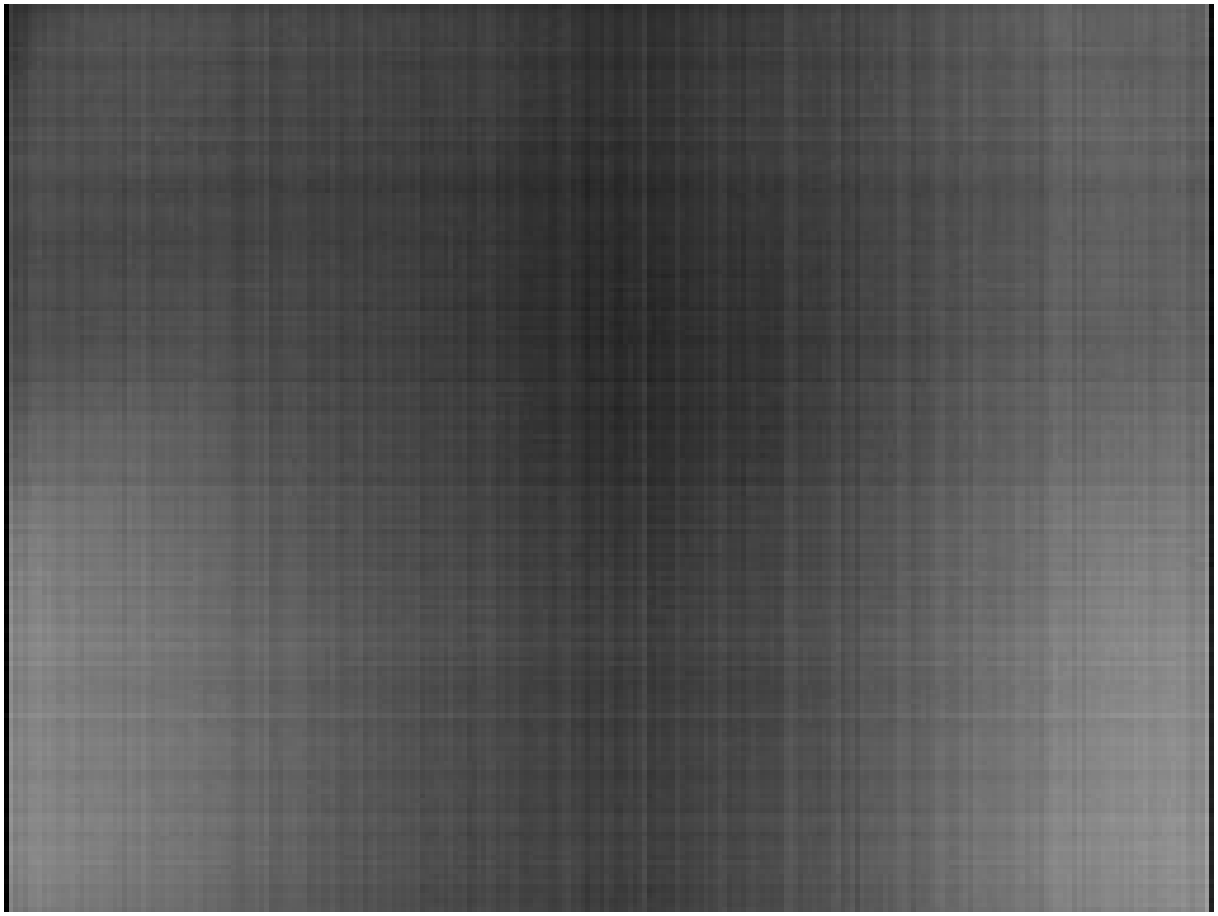
What about camera settings?

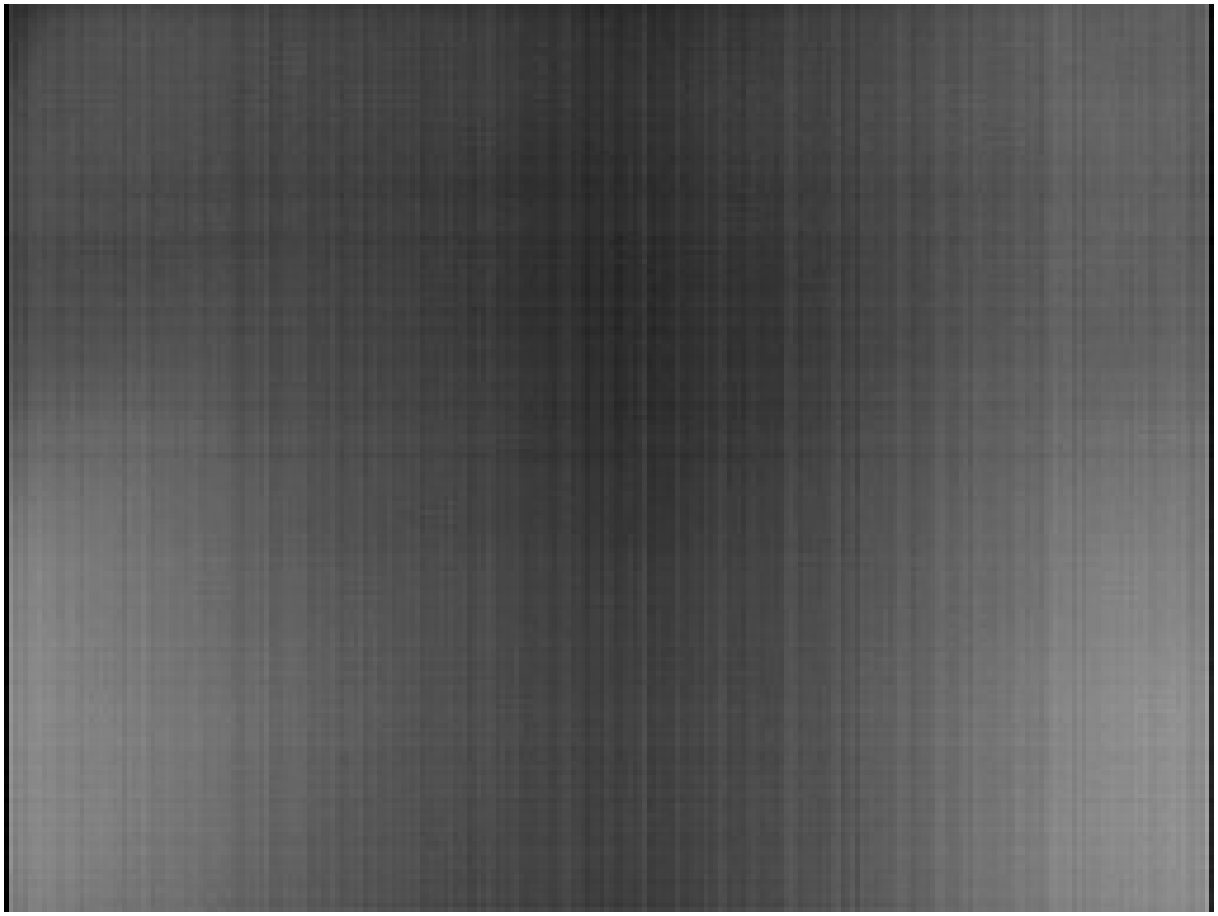
Unfortunately, dark frames depend on many camera settings: analog gain (ISO), exposure, other sensor settings like offset, black sun protection, PLR configuration and so on. Temperature is a variable as well.

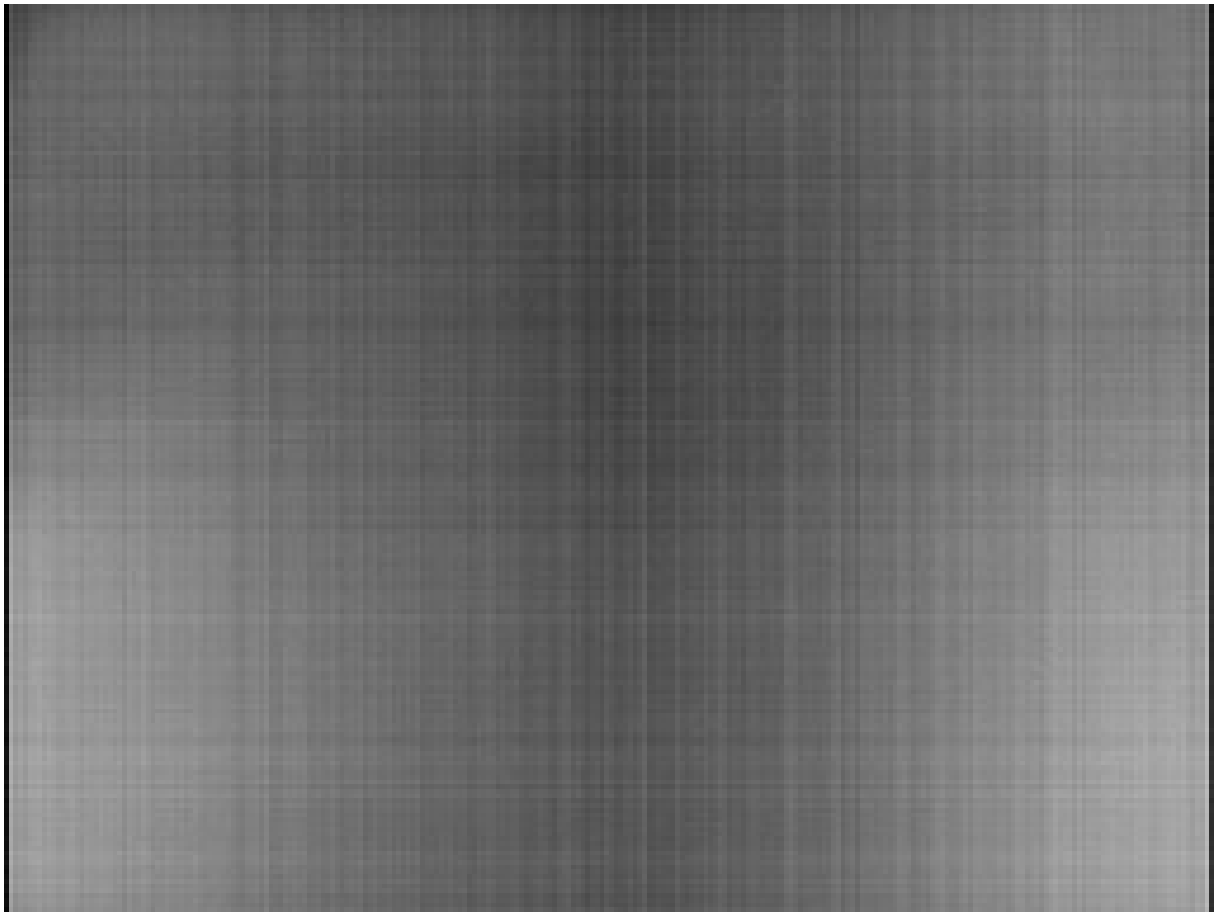
Luckily, the dependence on exposure appears to be linear, so we can take calibration frames at various exposures, combine them into a single dark frame, adjust it for the dark current and use it for the entire range of exposure settings (hopefully). We'll discuss that in the next section.

Dark current

Let's look at some dark frames: gain x1, exposures 1.2ms, 6ms and 78ms. Notice they get brighter as exposure increase.







The overall brightness in the dark frame changes with exposure in a linear fashion. We'll try to account for this in two ways: with a simple scalar value, and with a per-pixel correction.

The values in the black reference columns do not appear to compensate for the dark current, so we'll need to do it ourselves.

By identifying the dark current, we will be able to compute a dark frame that is applicable to any usual exposure time, but we are going to store only one or two reference frames for each gain. First reference frame will be called a bias frame (a zero-length exposure, that would contain only static black offsets), and the second reference frame, if used, will be called a dark current frame.

Dark current cannot be fully corrected because, while its stationary value can be measured and subtracted, it also introduces photon noise. Roger Clark explains it better:

"Dark current is temperature dependent and most modern CMOS digital cameras, circa 2008 and later have on sensor dark current subtraction, but while the dark current level is subtracted, the noise from the dark current still accumulates." - Source: http://www.clarkvision.com/reviews/how-to-interpret-reviews/

Simple correction

Experimentally, we have found the dark frame changes with exposure at roughly 0.065 digital units for each ms. This value is multiplied by analog gain. To find this value, take the dark frames at different exposure times (say 1...50 ms), then do a linear fit for the frame average (or median).

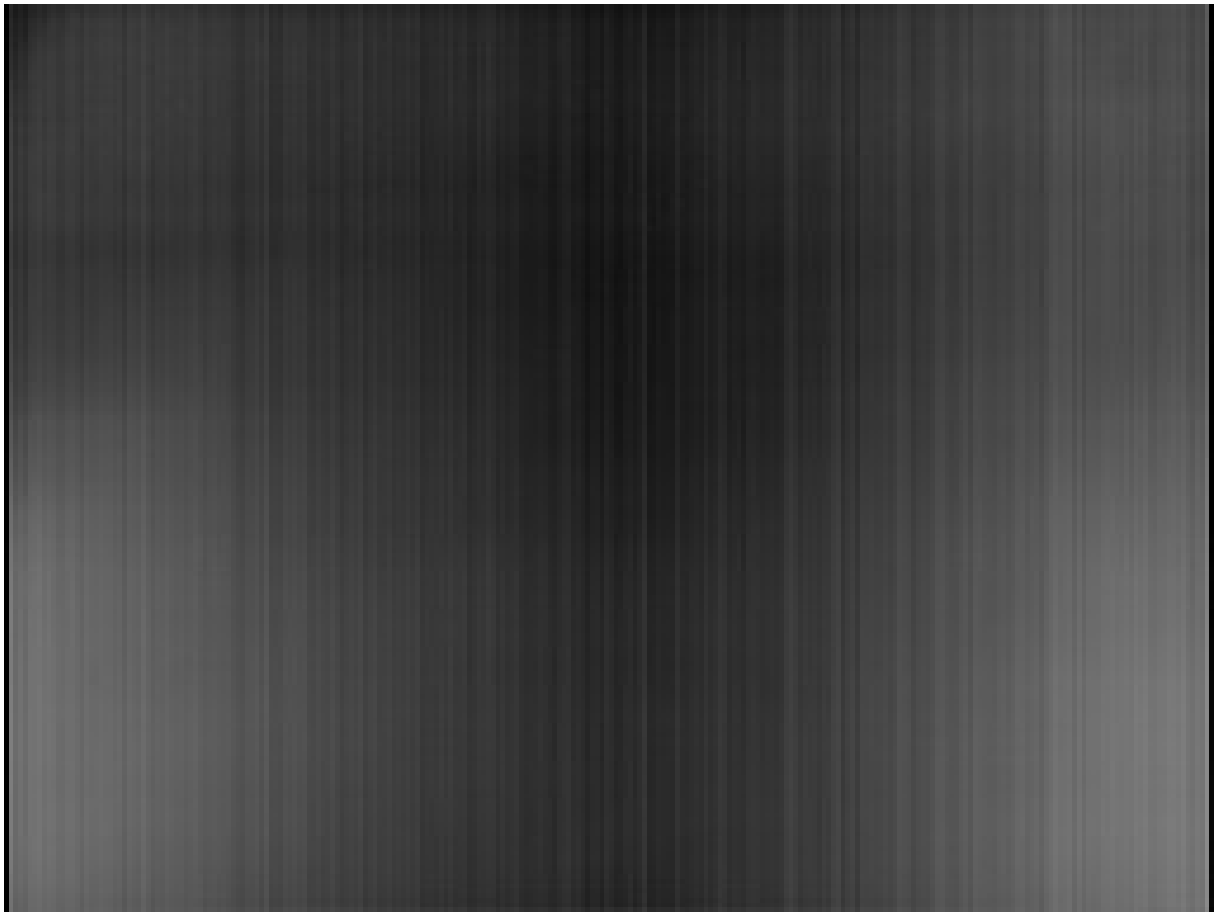
Command-line:

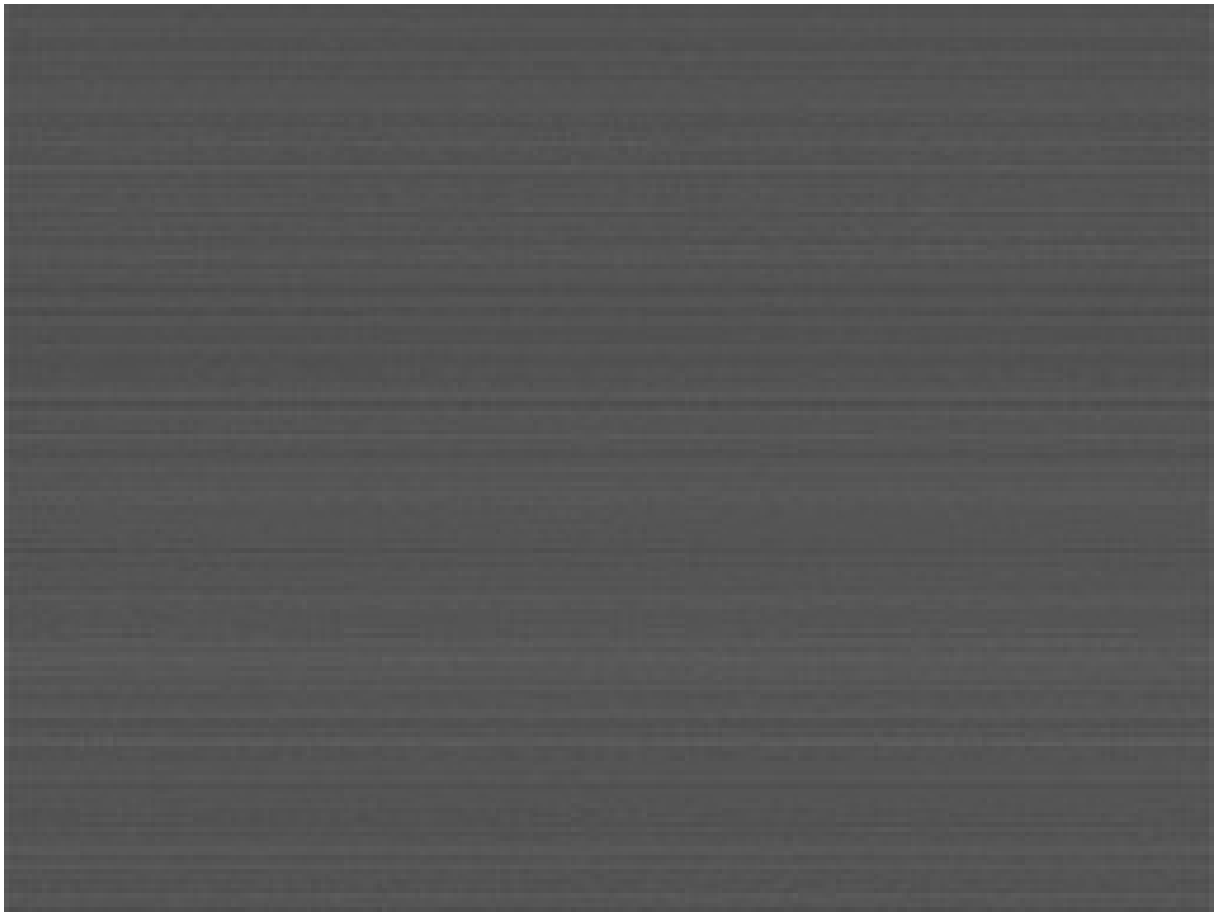
```
raw2dng *x1*.raw12 --calc- darkframe
```

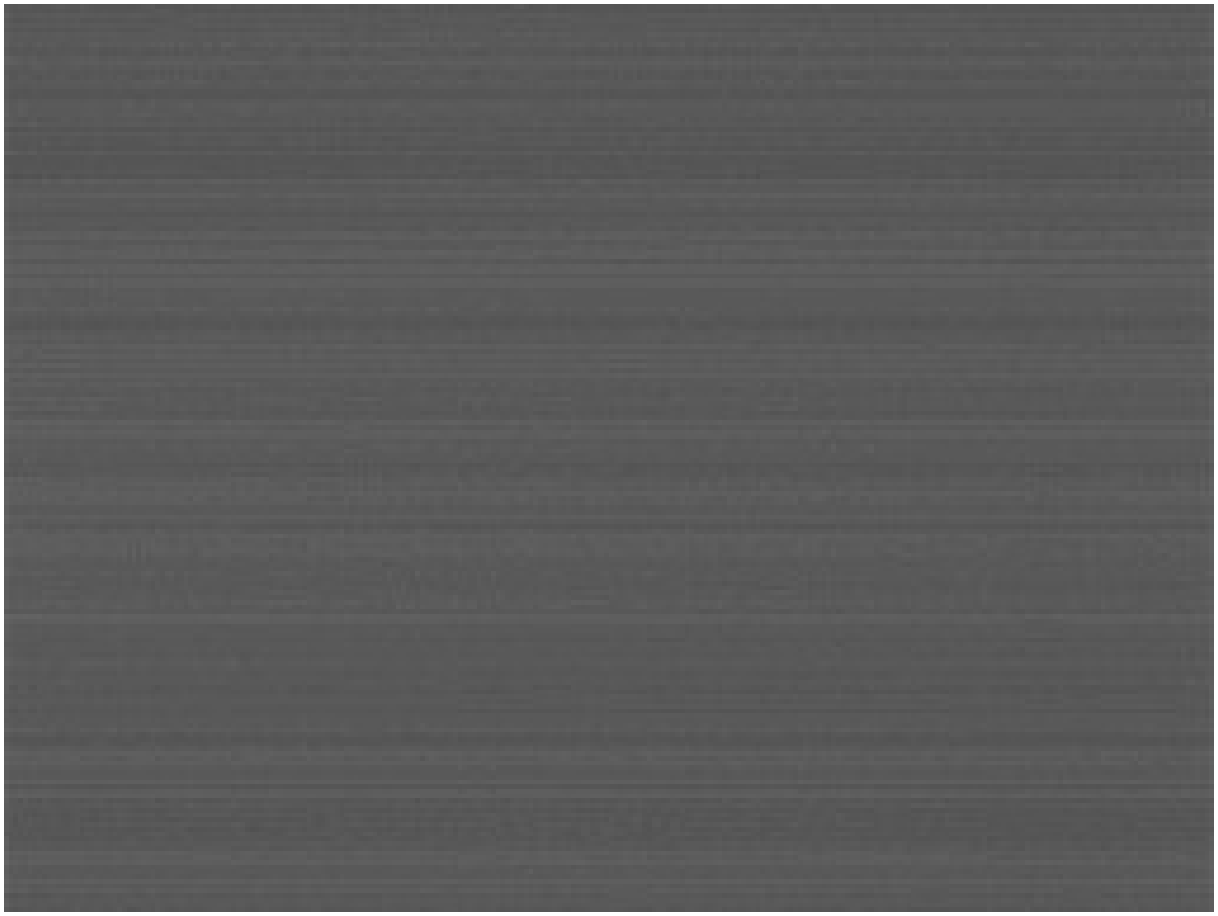
Example: a dark frame created from 256 exposures, between 1.2ms and 77ms, without using black reference columns. The image was adjusted (with a constant offset) to match a zero-length exposure, so calling it bias frame may be a good idea. If we use it to correct the individual dark frames, we should no longer see a variation in overall brightness.

```
Averaged 256 frames exposed from 1.19 to 76.79 ms. Dark current: 0.0653 DN/
ms
```

Image sequence: bias frame, single dark frames at exposures 1.2, 6 and 77 ms, corrected with the bias frame and the (scalar) dark current average. All files scaled to show a range of 60 DN, but the master dark frame has a different offset than the others.







11.1.1.2 Dark Current Non-uniformity Correction One common use of bias frames is for scaling dark frames. By subtracting a bias frame from a dark frame, you end up with a thermal frame. A thermal frame contains pixel values showing just the effect of dark current. Because dark current in any given pixel accumulates at a constant rate, a thermal frame allows you to predict with reasonable accuracy how much dark current there would be for different length exposures. However, given the opportunity, youre always better off taking dark frames that match the exposure times of your light frames.

http://qsimaging.com/ccd_noise_measure.htmlSource:

"Dark current non-uniformity is a noise that results from the fact that each pixel generates a slightly different amount of dark current. This noise can be eliminated by subtracting a dark reference frame from each image. The dark reference frame should be taken at the same temperature and with the same integration time as the image." - Source: https://www.photometrics.com/resources/learningzone/darkcurrent.php

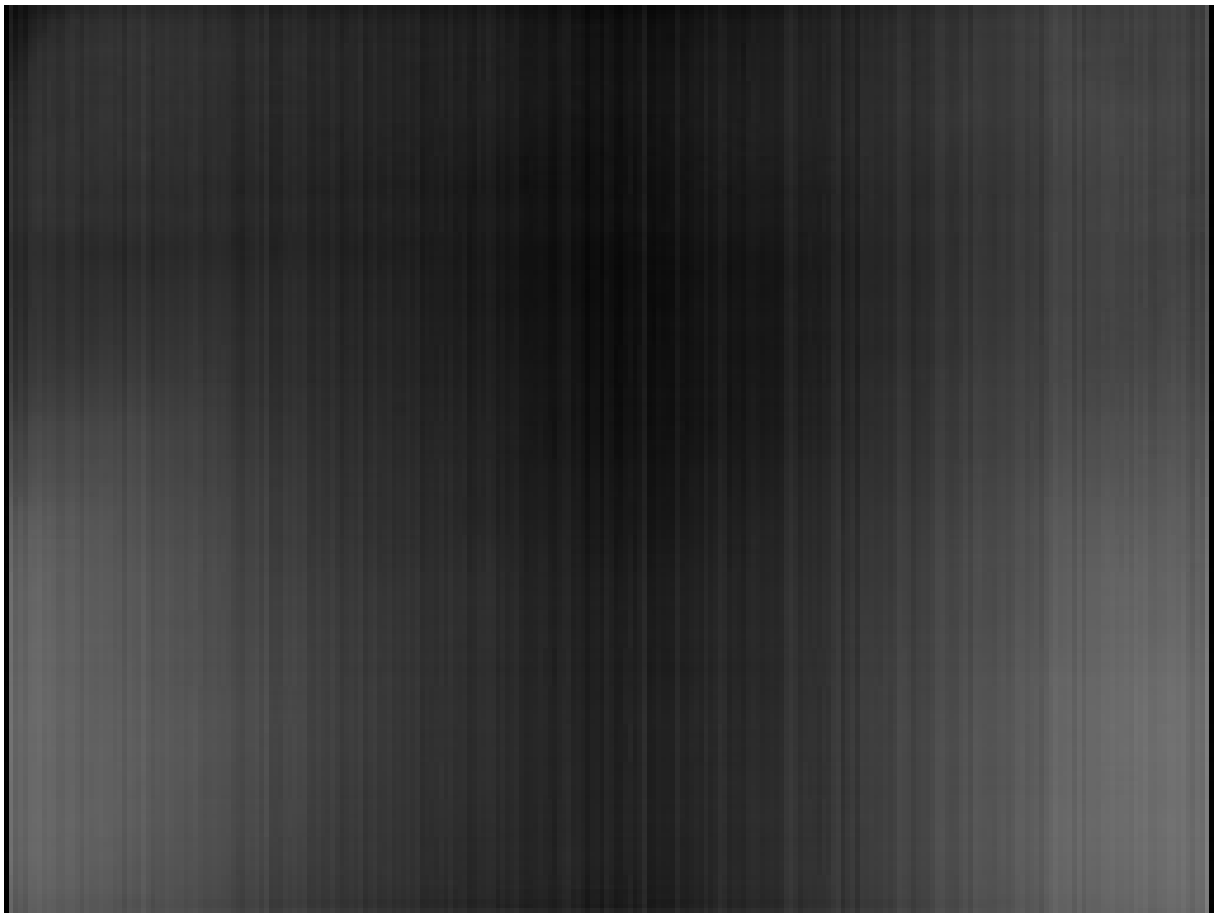
"Dark noise is not random; in fact, it is highly repeatable. A given photosite on a sensor will accumulate almost exactly the same amount of dark noise from one exposure to the next, as long as temperature and exposure duration do not vary."

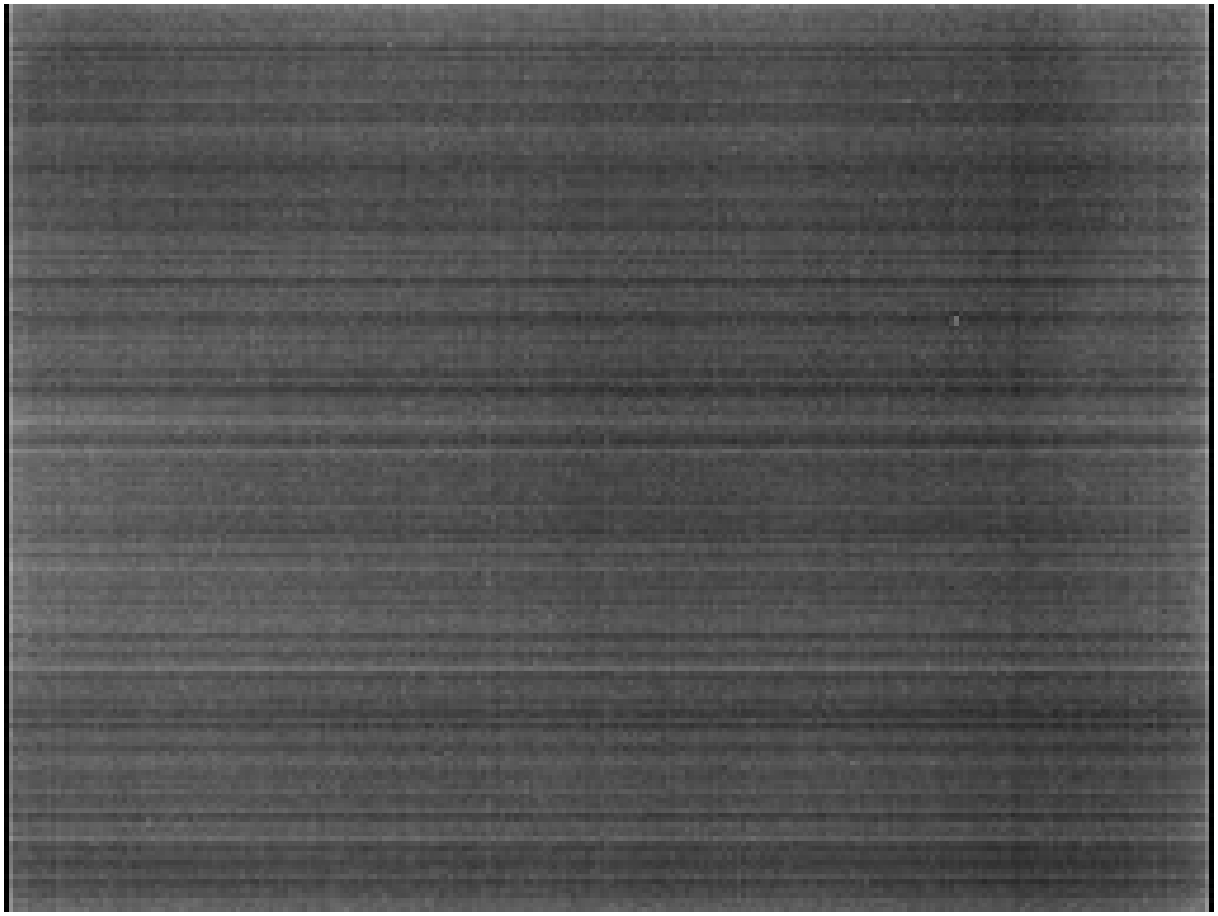
So, rather than using a single scalar value (0.06 dn/ms/gain) for all pixels, we can try finding the individual dark current for each pixel. Instead of doing a linear fit on the overall dark frame brightness (vs exposure), we will do the linear fit per pixel. We'll have to acquire a lot more dark frames to compute a good result, but it might be worth the trouble.

Command-line:

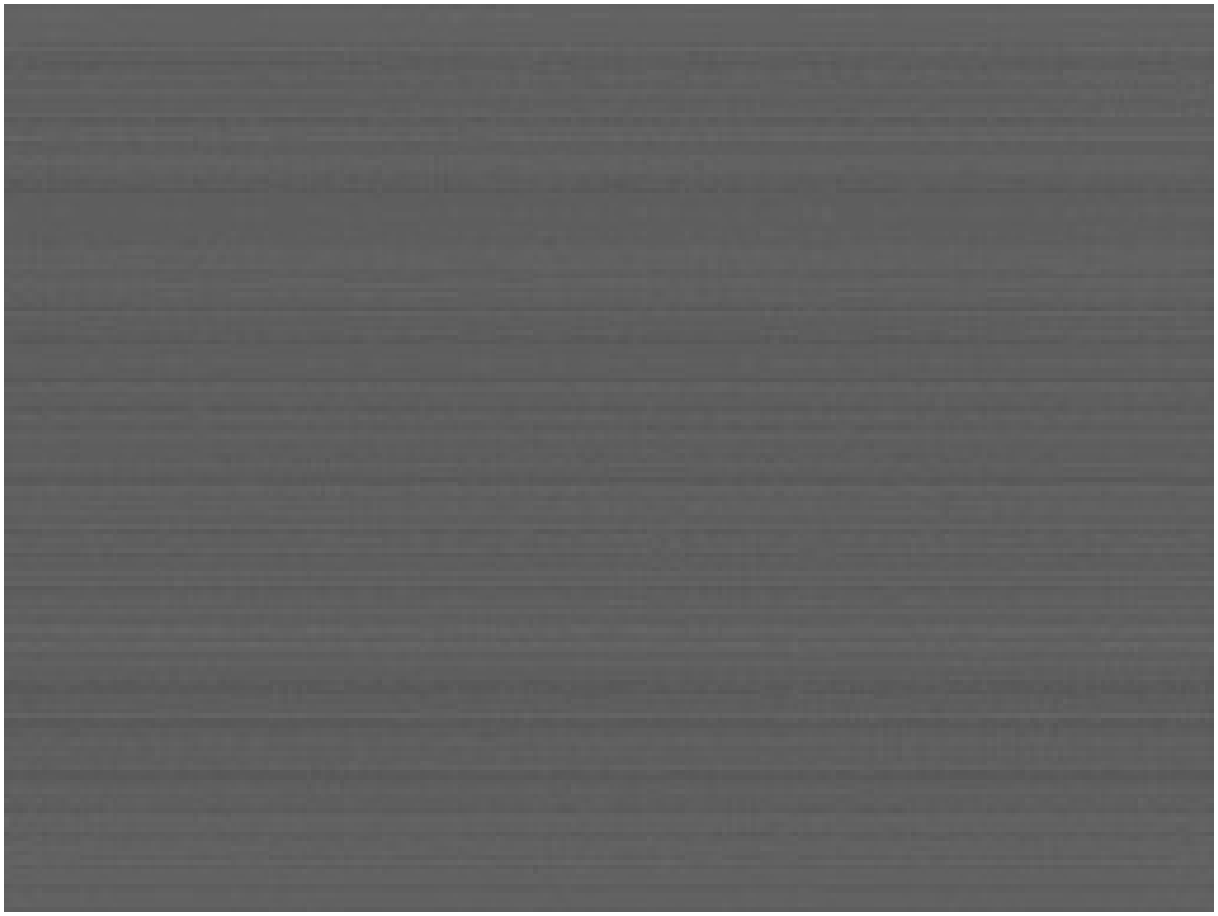
```
raw2dng *x1*.raw12 --calc- dcnuframe
```

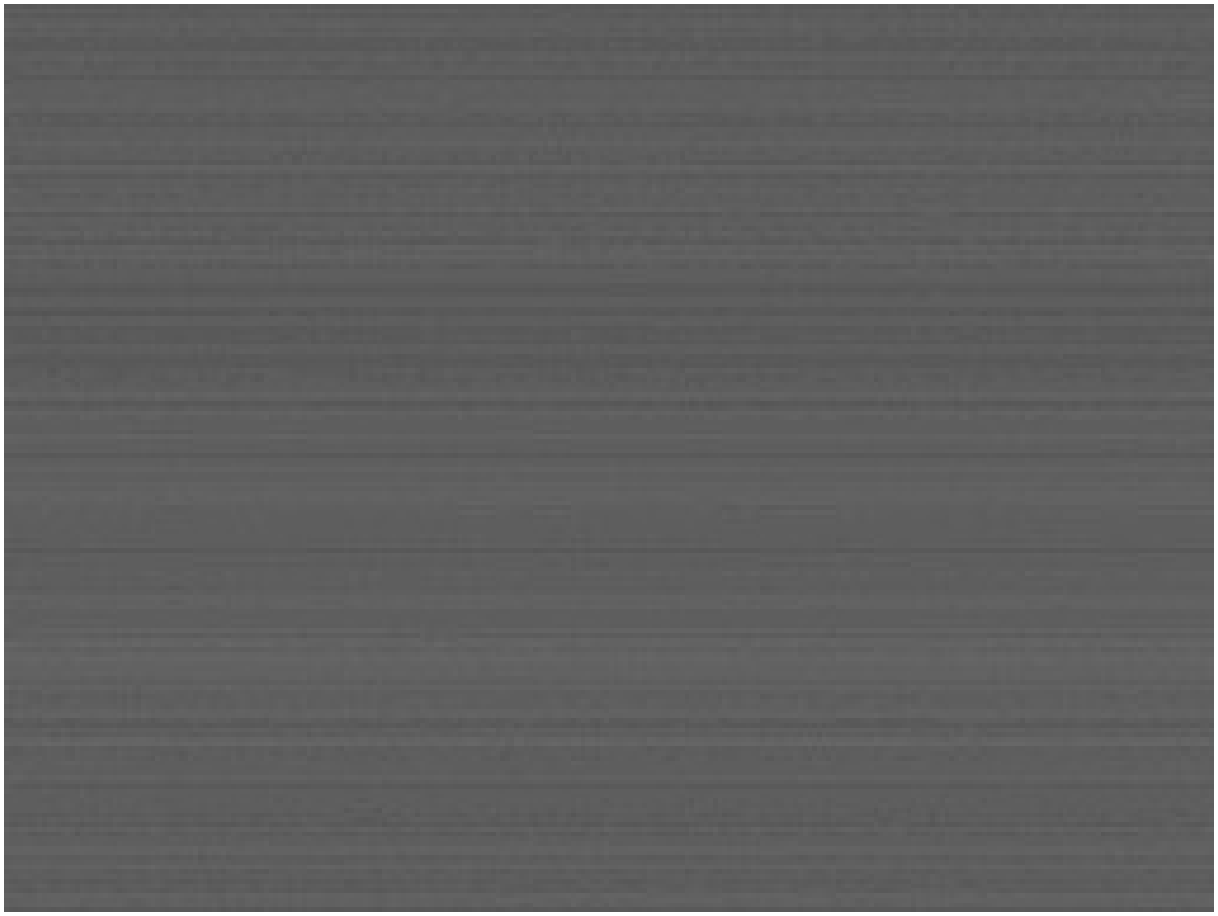
Example: bias frame (static offset) and dark current frame (exposure-dependent offset). Notice the bias frame looks quite similar to the previous one, but a little darker. The median value of the dark current frame is, unsurprisingly, 0.0645 DN/ms.

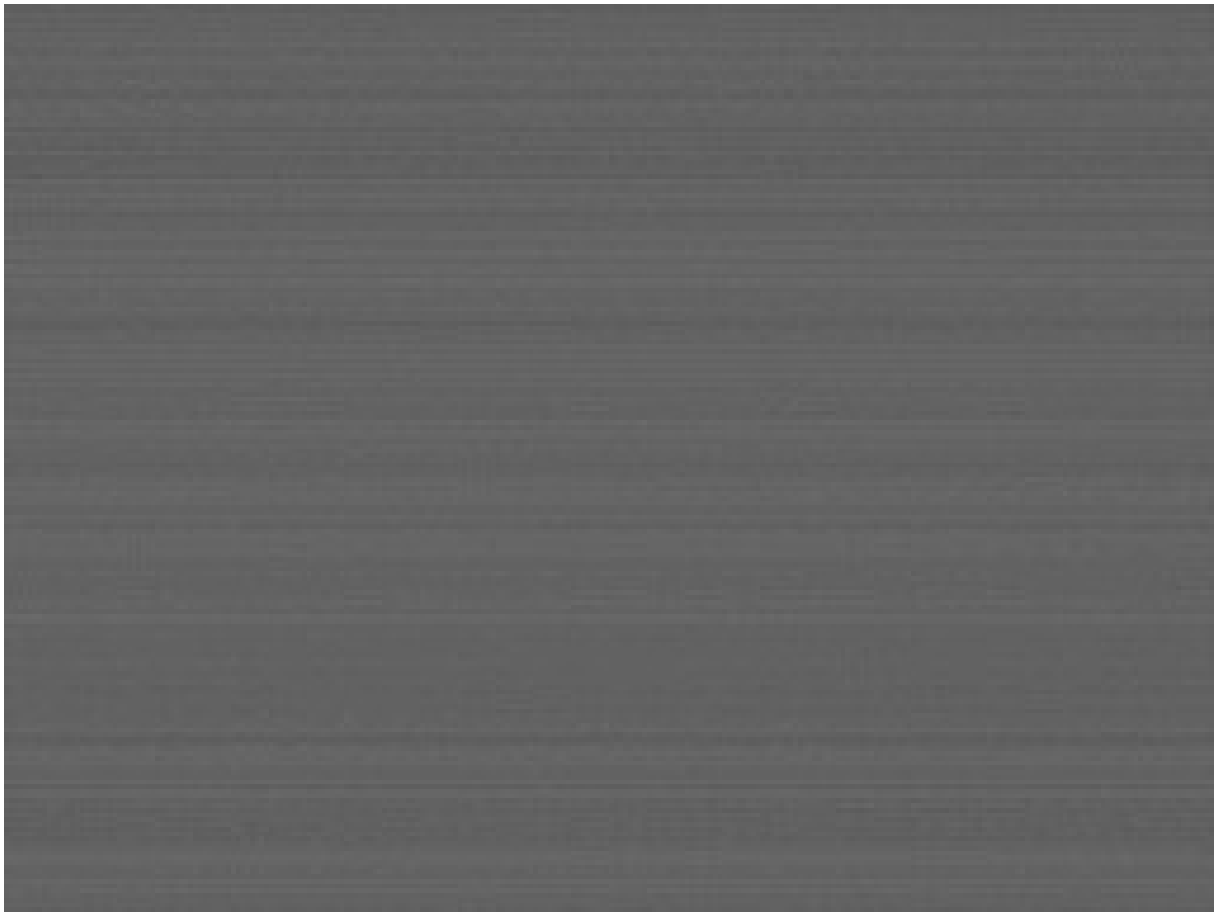




Individual dark frames (1.2, 6 and 77 ms) adjusted with dark current nonuniformity:







No obvious improvement in the test images, so why bother?

Let's correct all these 256 dark frames and check a few indicators: median, stdev, row noise and column noise.

Dark frame + scalar dark current:

Dark frame + dark current frame:

You may notice:

- Median (black level) variation: noticeable improvement with the second method.
- stdev (overall noise): minor improvement at extreme settings.
- Row noise: identical with both methods.
- Column noise: small improvement with the second method.

Indeed, the more complex method appears just a tiny bit better than the simpler one.

11.1.1.3 Dark Current Measurement From Hot Pixels A very interesting idea can be found here - https://www.photonics.com/Article.aspx?AID=44298 ... where hot pixels can be used to measure the

amount of dark current and scale it properly. This will probably account for changes in temperature, and may work at very long exposures without actually having to calibrate the camera in these conditions. Genius, if you ask me.

11.1.1.4 Black Reference Columns This sensor has 8+8 columns that can be used for calibrating the black levels; they are also useful for reducing the dynamic row noise.

Experimentally, we have noticed that odd rows have slightly different statistics (noise level, offset, gain), compared to even rows. This happens in both the black columns and the active area, and it may indicate two parallel circuits used for readout, each having slightly different electrical response.

Therefore, it may be wise to process the black columns for odd and even rows separately, which should already fix some issues like static row noise, or the need for green equilibration.

In raw2dng, this correction is enabled by default, as long as you use a dark frame. You can turn it off with `–no-blackcol`, if you want.

11.1.1.5 Black Level The sensor has two registers that can be used to adjust the black level: one for odd rows, another for even rows. This confirms our finding about two parallel readout circuits.

You might be tempted to adjust the black level to 0 (like Nikon does). Please don't. Here's why:

- If you adjust the offset until the black level becomes roughly 0, you will clip all the data below this level. Good luck subtracting a dark frame after that.
- Even if you change the level after doing all the black corrections, you may still have useful data below zero. You will need it when stacking multiple frames, or when doing noise reduction.

Just FYI, there is a hack for Nikon cameras that moves the black level above 0. See https://landingfield.wordpress.com/2014/0
nikon-dslr-black-point-hack-for-astrophotography/

... Sample photos can be found here or here.
We recommend setting the offset so that only a few isolated pixels (if any) reach the value of 0. A black offset of 2047 (registers 87/88) is a good choice. You won't lose any dynamic range by doing that.

On most recent Canon DSLRs, black level is 2048. That's a little on the large side, but it's a good thing. Some may argue that you may lose 2 stops or more of dynamic range by doing that (reading - https://www.reddit.com/r/photography/comments/3q4tnz/how_to_correctl_push_5_stops_with_canon/cwcbyb5/), but this is wrong. On Canons, the raw output is 14-bit, so by setting the offset to 2048 instead of 0, the useful range will be "just" log2(16384-2048) = 13.8 bits, instead of 14. So, yeah, you lose 0.2 bits from the ADC range.

With the black offset of 2047 on the CMV12K, the black level ends up at around 150, so you lose a whooping 0.05 bits from the 12-bit range.

Note: for easier processing, raw2dng shifts the raw data in order to fix the black level at 128.

Note also: Row noise correction from black columns detailed in 11.2 Pattern Noise.

11.1.2 Checking Black Level

You may wonder: after all these corrections, did we get the right black level? Can we render shadow detail correctly?

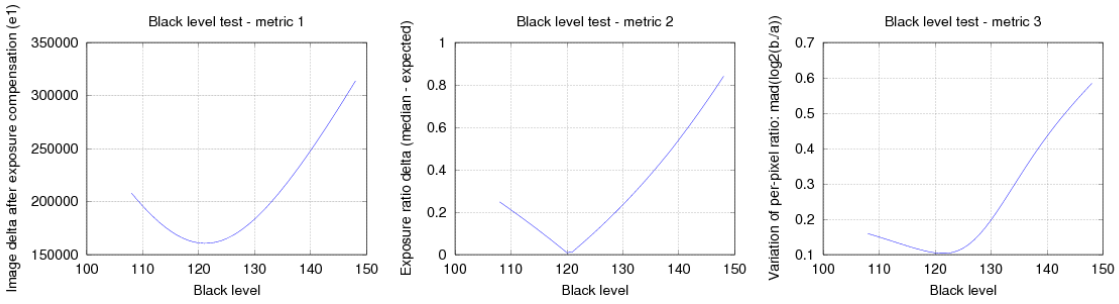
A possible criteria for checking: if we have two images of the same scene, taken with different exposures, we should be able to match those in postprocessing by simply dragging the exposure slider. For example, if we have one image at 5ms and another one at 20ms, we would set the exposure to +2 EV on the first image - the results should be pretty much identical, except for noise (and clipped highlights, if we weren't careful with the exposure).

Of course, that will work once we know the sensor Response Curves (See 11.5).

To check how well the images can be exposure-compensated, we could evaluate (and minimize) one of those error metrics:

```
a  = dark\_image - black\_level;
b  = bright\_image - black\_level;
e1 = norm(a*expo\_ratio - b);
e2 = abs(median((b ./ a)(:)) - expo\_ratio)
e3 = mad(log2((b ./ a)(:)));
```

First metric checks the difference between the two images, where the darkest one was adjusted by scaling (gain) to match the brightest one. Second metric computes the ratio between the two images at each pixel, and checks its median value vs the expected value. Third one checks the variation of per-pixel ratios between the two images, in stops, ignoring the expected value - this metric could be useful if we suspect the exposure controls may not be accurate.

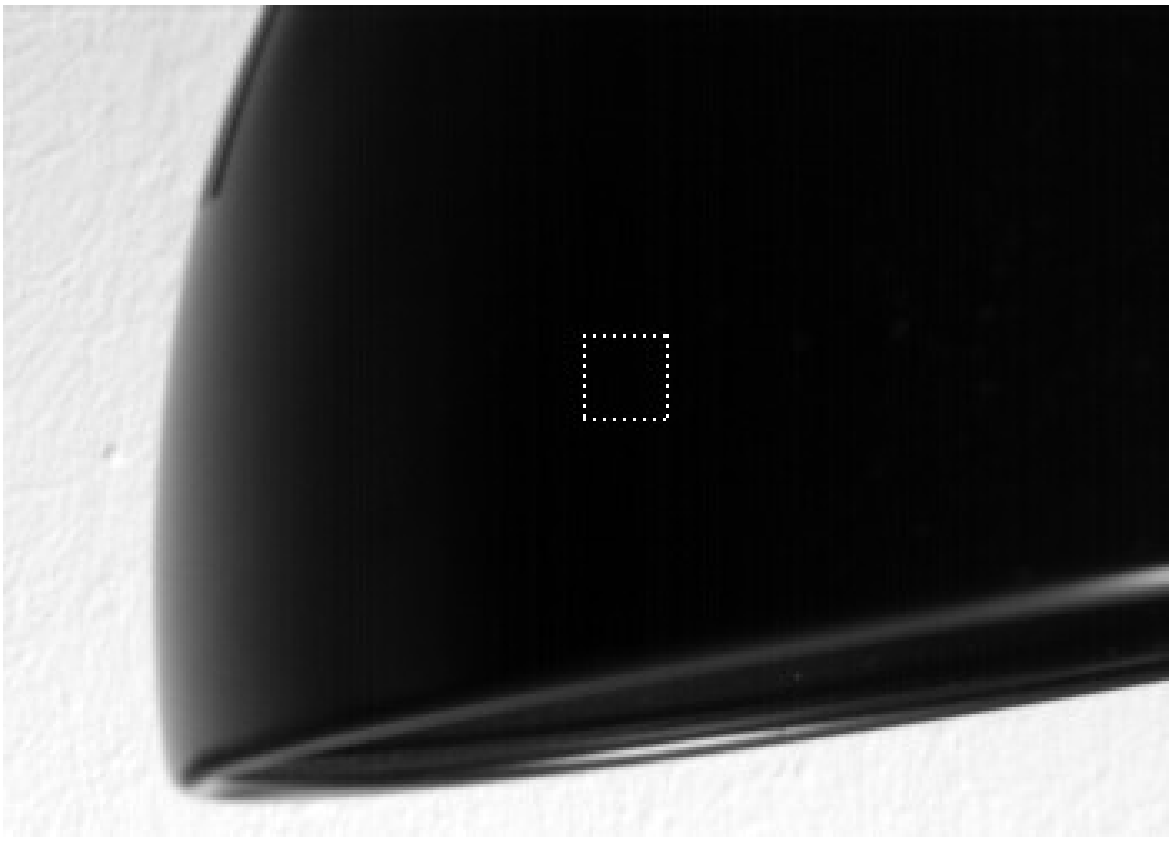


With all 3 methods, minimization indicates a black level of around 120-122 (expected 128), so there's still something missing with our calibration. This is confirmed by missing details in very dark areas (crushed blacks) on some sample images.

On the same test image, the response curve estimated from the grayscale IT8 reference data indicates a black level of 129.
On the same test image, the response curve estimated with the Robertson02 algorithm indicates a black level of 124 (See 11.5).

Who is right?

The sensor also has a strange behavior - something we have called the "Black Hole" anomaly: in very dark areas, sensor output decreases with exposure time. This might give a clue for solving this mystery.



11.1.3 Calibration Pipeline

- Use black reference columns to find the black levels for odd and even rows.
- Subtract dark offset and dark current.
- Use variations in black reference columns to reduce row noise.

These operations should be simple enough to be implemented in FPGA, as real-time corrections.

11.1.4 Calibration Procedure

Quick calibration

Acquire 16 dark frames, gain x1, exposures between 1 and 50 ms, save them under the 'darkframes' subdirectory, then run:

```
raw2dng darkframes/*x1*.raw12 --calc-darkframe --swap- lines
```

Result: darkframe-x1.pgm.

Repeat for other gains if needed.

Accurate calibration, with dark current nonuniformity
Acquire 256 dark frames, gain x1, exposures between 1 and 64 ms in linear increments (4 images at each setting), save them under the 'darkframes' subdirectory, then run:

```
raw2dng darkframes/*x1*.raw12 --calc-dcnuframe --swap- lines
```

Result: darkframe-x1.pgm and dcnuframe-x1.pgm.

Repeat for other gains if needed.

Checking the calibration

You can verify the calibration by rendering the same individual dark frames, this time corrected, to check the residuals. Or, even better, acquire a new set of dark frames at the same settings, and check those instead:

```
raw2dng darkframes/*x1*.raw12 --check-darkframe --swap- lines
```

You will see the average value, pixel noise and row/column noise levels (both absolute and relative to pixel noise) for each dark frame. Example:

```
Average : 127.48 Pixel noise : 2.48 Row noise : 0.61 (24.6%) Col noise : 0.05
(2.2%)
```

Using the reference frames to correct raw12 files.

1. Place the calibration files in the working directory.
2. Make sure your .raw12 images contain a metadata block.
3. raw2dng will recognize the dark frames and use them for correcting your image.

```
cp /path/to/darkframes/*-x[1-4].pgm . raw2dng *.raw12 --swap- lines
```

Note: you need –swap-lines as an workaround for an old bug introduced in the Beta, but wasn't fixed yet in the FPGA.

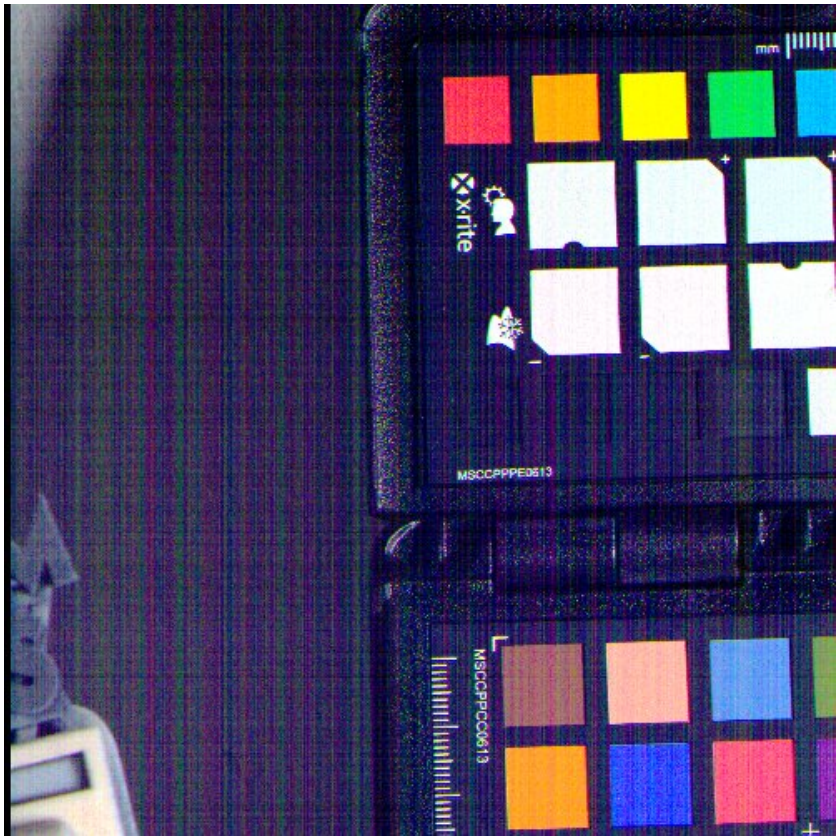
This is not exactly useful when dealing with multiple folders, so until we'll have a better way to organize the calibration frames, you may try an alternative workflow:

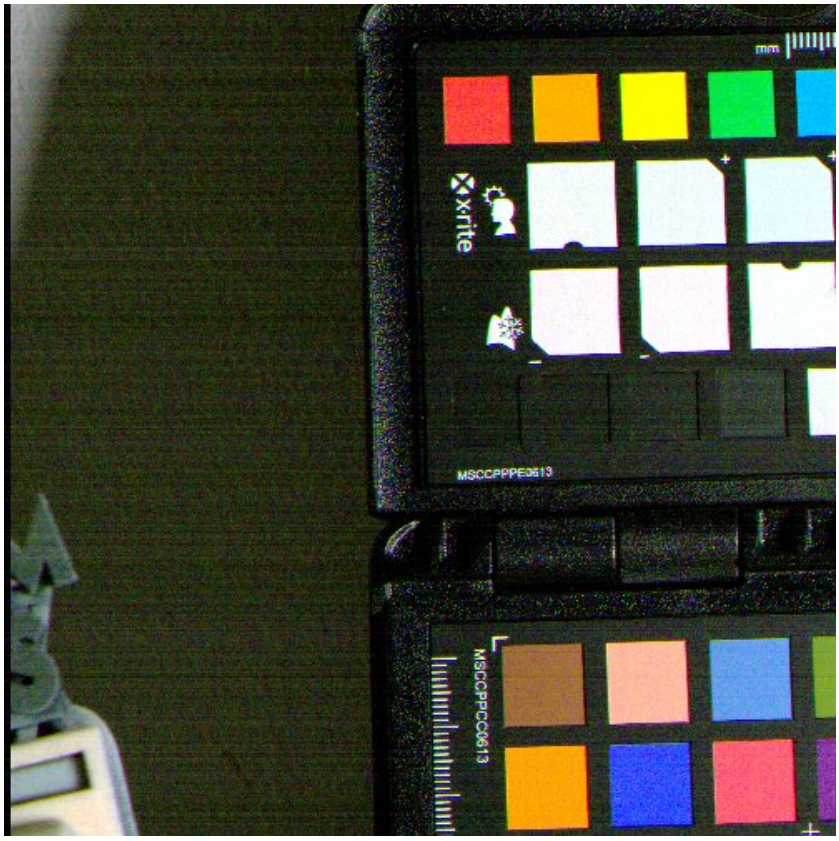
1. Place the calibration files in some directory (let's call it "calibration directory")
. 2. Make sure your .raw12 images contain a metadata block.
3. Use paths when passing input files raw2dng (the output files will be saved in the same directory as the input file).

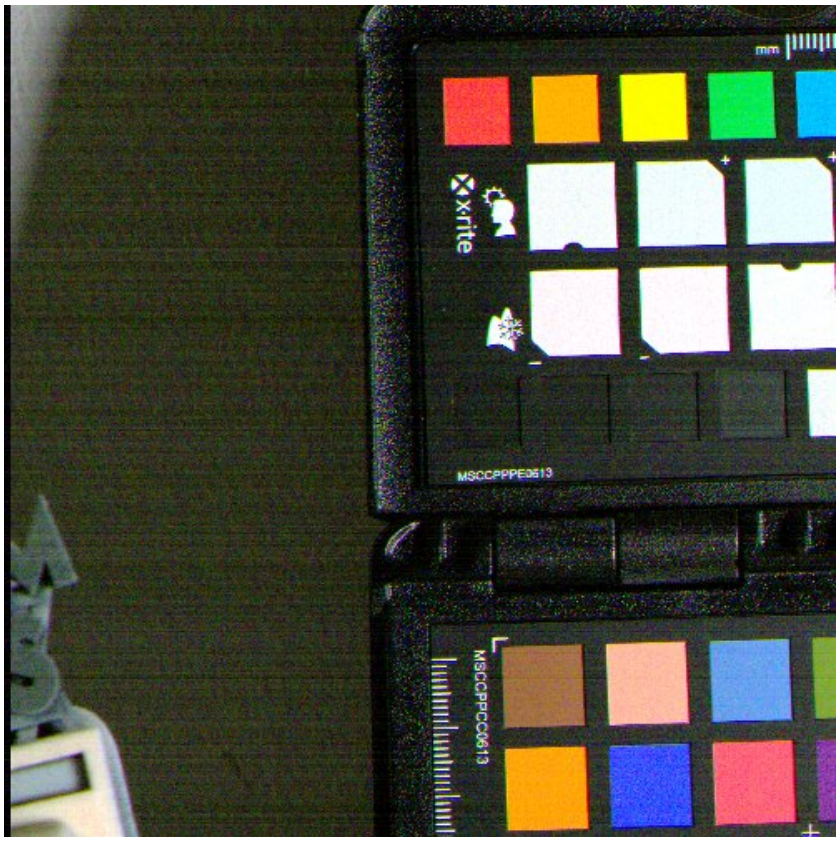
```
cd /path/to/darkframes ls *.pgm darkframe-x1.pgm dcnuframe-x1.pgm ... raw2dng /path/to/
lines
```

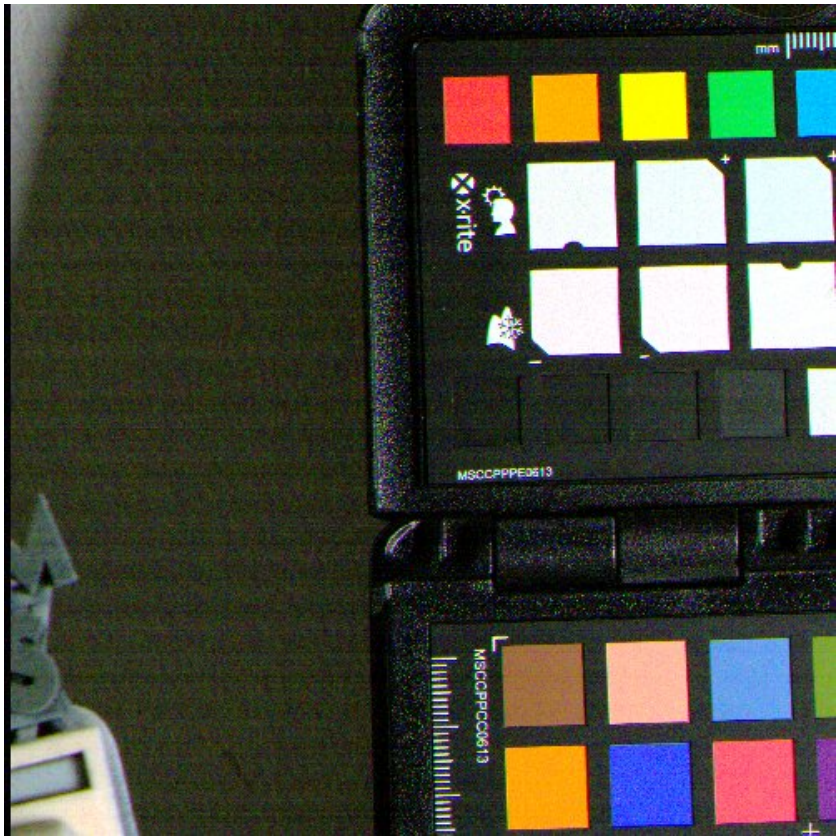
Example
Showing half-res image crops pushed by 4 stops (ufraw-batch –wb=auto –exposure=4 –shrink=2).
- Top left: raw sensor data (adjusted black level manually).
- Top right: corrected with dark frame, scalar dark current, no black columns.
- Bottom left: corrected with dark frame, dark current frame, no black columns.
- Bottom right: corrected with dark frame, dark current frame, black columns enabled.

Note: in the raw data, even and odd rows have different black offsets; that's why we have wrong colors.




mm
xrite
MSCCPPPE0613
MSCCPPCC0613


mm
xrite
MSCCPPPE0613
MSCCPPCC0613



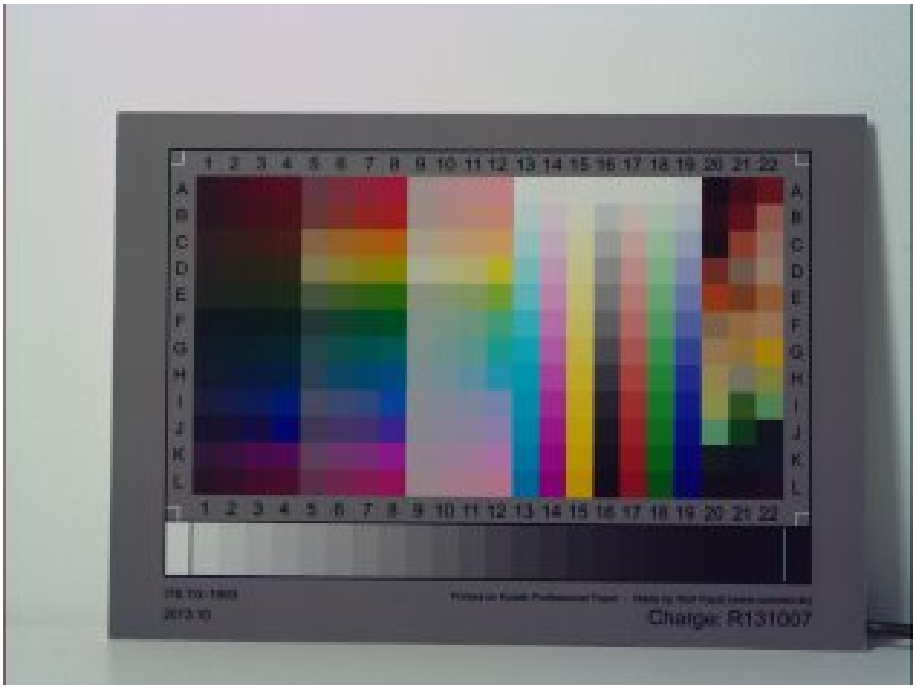
11.2 Pattern Noise

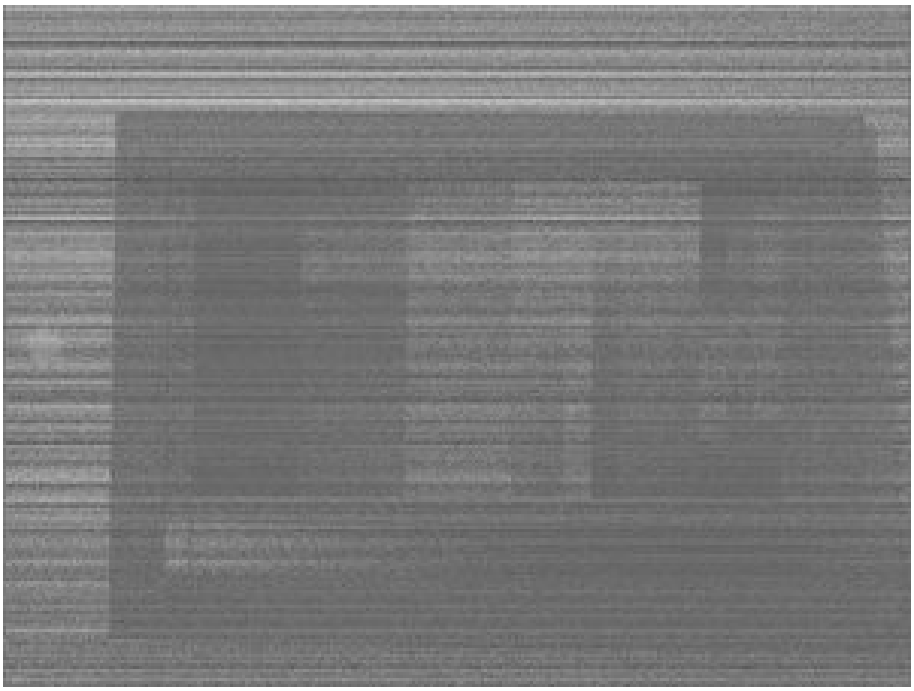
General info about pattern noise can be found here - http://theory.uchicago.edu/ ejm/pix/20d/tests/noise/#patternnoise

The CMV12000 sensor suffers from dynamic row noise.

That means, a scalar offset gets added to each row. The offset is not correlated between different frames, so we can't remove it using a calibration frame (dark frame or whatever).

One can observe this noise by looking at the difference between two images taken at identical settings. There are two main components that appear obvious in such a difference frame: random noise (per pixel, increases on brighter pixels) and row noise (per line).





11.2.1 Correction Methods

There are two ways to deal with this noise, after performing Black Calibration (See 11.1)

- Use info from black reference columns to reduce dynamic row noise without guessing anything (fast, can be implemented in real-time, see Raw Preprocessing).
- Use denoising techniques to reduce the remaining row noise (the guesswork part, slow).

Reducing row noise using black reference columns

See http://github.com/apertus-open-source-cinema/misc-tools-utilities/commit/48de47b2a544dc32bbd5a8fd7701bb44a31ea8624053a553f49c0036b4d31282e58b2fR301
The application note AN01 from CMOSIS says:

"The noise is also present in the black reference columns (8 left and 8 right), so when enabled (reg 89[15] = 1), these can be used for row noise correction by for example making a relative row profile of these black columns and subtract this from the image."

However, simply subtracting each row average of the black columns from our image is not going to work. Here's why:

Kalman filter theory: http://robocup.mi.fu-berlin.de/buch/kalman.pdf

From page 3, if we know how noisy our estimations are, the optimal weights are inversely proportional with the noise variances:

```
x_optimal = (x1 * var(x2) + x2 * var(x1)) / (var(x1) + var(x2 ))
```

Here, let's say R = x1 is row noise (stdev = 1.6 at gain=x1) and x2 is black column noise.

```
R = x1
B = mean(black\_col') = R + x2 =>  x2 = B - R
x2 can be estimated as mean(black\_col') - mean(active\_area')
stdev(x2) = 1.3.
```

We want to find k that minimizes var(R - k*B).

```
var(R - k*B) = var(x1 * (1-k) - x2 * k),
=> k = var(x1)) / (var(x1) + var(x2).
```

In particular, for gain = x1, k = $1.6\hat{2}$ / ($1.6\hat{2}$ + $1.3\hat{2}$) = 0.6.

So, we don't have to simply subtract the black columns. Rather, we'll subtract the static offset (median value) first, and then, we'll subtract the remaining variations multiplied by 0.6 at gain=x1.

Things get a little more complex because the static offset is different on odd and even rows, and it also appears to change from the left side to right side of the frame. More details on the **Raw Preprocessing** page.

Fixed frequency perturbation in black columns

A closer look at the frequency spectrum of the black columns, compared to the spectrum of the row noise from a dark frame, revealed a strong fixed-frequency component present only in the black columns. Attempting to fix row noise with the above procedure would introduce some of this fixed frequency component in the main image as well.

In the example image from below, this component has a frequency of 1/41.27 pixels-1, with an amplitude of 1.14 DN. The value is different in other test images, and appears to be consistent in the images taken during the same experiment. It doesn't change with exposure time. Cause is unknown.

TODO: detailed analysis, FFT graphs...

We'll attempt to filter out this perturbation from the black columns before using them for reducing row noise.

Good news: after removing this perturbation, the optimal black columns multiplier increases to about 0.8 :)

Using nearby rows to reduce the row noise even more

Maybe some nearby rows could offer some hints about the row offset on the line being analyzed?

Looks like yes. Trying to write the row noise as a linear combination of row averages from black columns, using linear regression, not just the ones from the same row, but also from nearby rows, and discarding very small coefficients, gives the following filter:

```
(y % 2)
   ?
     black\_col[y-1] *  0.24 +
     black\_col[y]   *  0.63
   :
     black\_col[y]   *  0.38 +
     black\_col[y+1] *  0.43
```

What's interesting: on even rows, row noise (in the active area) depends on the black column average from the same row, but also on the black column average from the previous row. But on odd rows, it depends on the black column average from the same row and the next row. This probably means they are processed in pairs, and there is some common perturbation that affects both of them.

This gives a minor improvement (0.6 -¿ 0.5).

Exploiting differences in green channels
The two green channels are expected to be nearly identical, except for high-frequency details like sharp edges. That means, if we take the median difference on each row, we expect to get zero. In practice, we get some nonzero values. We could try to see if there is any correlation between those green channel differences and our row noise.

Let's extract only the green channels from an image: the result will be W x H/2. Let's define green_delta(lag) = row_median(green - circshift(green,-lag)). That means, from each green row, subtract some nearby green row, and take the median difference. We'll compute this at lags -2, -1, 1, 2, mapped to array indices 0, 1, 2, 3.

Linear regression gives an ugly FIR filter that looks like this:

```
(y % 2)
```

```
        ?
          black\_col[y-2]    * 0.17 +
          black\_col[y-1]    * 0.14 +
          black\_col[y+0]    * 0.17 +
          black\_col[y+1]    * 0.16 +
          black\_col[y+2]    * 0.14 +
          green\_delta[0][y] * 0.22 +
          green\_delta[1][y] * 0.31 +
          green\_delta[2][y] * 0.38
        :
          black\_col[y-2]    * 0.12 +
          black\_col[y-1]    * 0.13 +
          black\_col[y+0]    * 0.14 +
          black\_col[y+1]    * 0.14 +
          black\_col[y+2]    * 0.12 +
          black\_col[y+3]    * 0.12 +
          green\_delta[0][y] * 0.33 +
          green\_delta[3][y] * 0.32
```

As weird as it looks, it seems to work!

Improvement (over the "Kalman" scaling of black columns): 0.6 -¿ 0.3.

Reducing the remaining row noise by image filtering

Basic algorithm:

1. Filter the image with an edge-aware vertical blur (bilateral filter on pixels from the same column)
2. Subtract the blurred image; the residuals will reveal the row noise (see example)
3. Mask out highlights and strong edges
4. Take the median value from each row of the residuals image
5. Subtract these values from each row of the original image

Source: https://github.com/apertus-open-source-cinema/misc-tools-utilities/blob/master/raw2dng/patternnoise.c

11.2.2 Usage

The methods discussed here are implemented here = https://github.com/apertus-open-source-cinema/misc-tools-utilities/tree/master/raw2dng

- Black reference columns are used by default, as long as you use a dark frame (since this method is fast and has no side effects)
- To reduce the remaining row noise, use `raw2dng –fixrn`
- If the image also suffers from column noise, use `raw2dng –fixpn`

Troubleshooting or checking the effectiveness of each step:
- Disable row noise reduction from black columns, but use the static offsets: `–no-blackcol-rn`
- Disable fixed frequency correction for black columns: `–no-blackcol-ff`
- Disable black reference columns completely: `–no-blackcol` (you need to compute new darkframes if you

use this)

Tip: the algorithm for filtering row noise is also available in MLVFS, so you can use it on MLV videos (recorded with Magic Lantern) as well. See - http://www.magiclantern.fm/forum/index.php?topic=13152

11.2.3 Tests on a dark frame

Checking a single 10ms dark frame, corrected using 256 exposures between 1.2ms and 77ms, various settings. Showing only noise measurements.

```
# raw data without any processing
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --no-darkframe --check-darkf
  Average     : 190.35
  Pixel noise : 7.14
  Row noise   : 5.29 (74.1%)
  Col noise   : 5.41 (75.7%)



# disable black columns, simple darkframe
raw2dng *x1*.raw12 --calc-darkframe --no-blackcol --swap-lines
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --no-blackcol --check-darkfr
  Average     : 140.33
  Pixel noise : 3.08
  Row noise   : 1.33 (43.0%)
  Col noise   : 0.09 (3.0%)



# disable black columns, enable dark current frame
raw2dng *x1*.raw12 --calc-dcnuframe --no-blackcol --swap-lines
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --no-blackcol --check-darkfr
  Average     : 142.46
  Pixel noise : 3.06
  Row noise   : 1.32 (43.1%)
  Col noise   : 0.08 (2.5%)



# dark current frame, use only static offsets from black columns
raw2dng *x1*.raw12 --calc-dcnuframe --swap-lines
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --no-blackcol-rn --check-dar
  Average     : 127.46
  Pixel noise : 3.06
  Row noise   : 1.39 (45.4%)
  Col noise   : 0.10 (3.2%)



# dark current frame, enable black columns, don't remove the fixed frequency component from b
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --no-blackcol-ff --check-dar
  Average     : 127.46
```

```
  Pixel noise : 2.90
  Row noise   : 0.82 (28.2%)
  Col noise   : 0.10 (3.3%)
```

```
# dark current frame, enable black columns, remove the fixed frequency component (default set
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --check-darkframe
  Average     : 127.46
  Pixel noise : 2.85
  Row noise   : 0.61 (21.4%)
  Col noise   : 0.10 (3.4%)
```

```
# also enable --fixrn (aggressive correction)
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --fixrn --check-darkframe
  Average     : 127.42
  Pixel noise : 2.79
  Row noise   : 0.12 (4.1%)
  Col noise   : 0.10 (3.5%)
```

So, row noise was reduced from 1.33 (after dark frame correction) to 0.61 after using the black reference columns.

If it's still noticeable, –fixrn should bring it to very low levels, unless you have strong horizontal lines in the image, which might trick the algorithm.

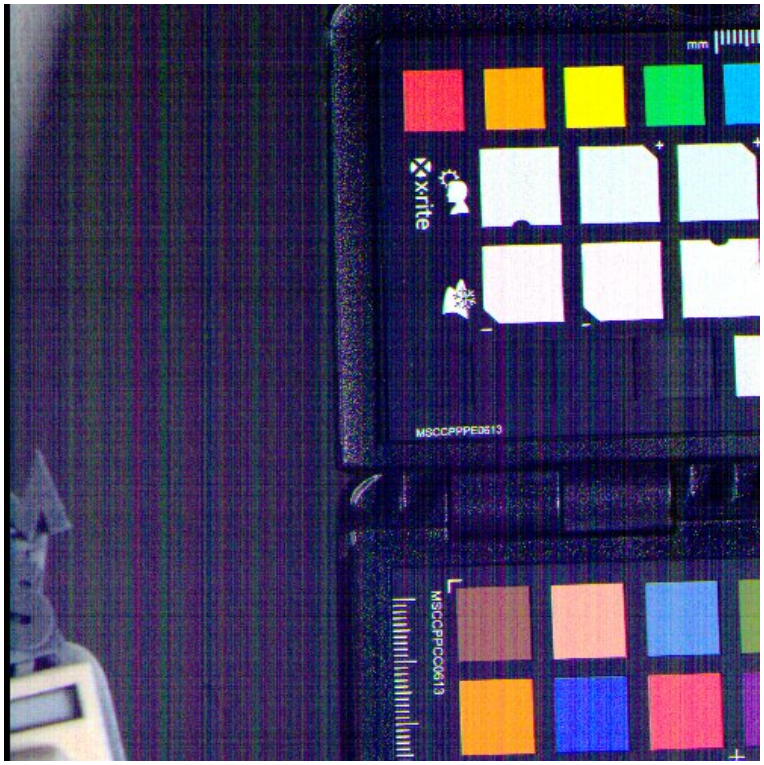
Some experimental row noise filters:

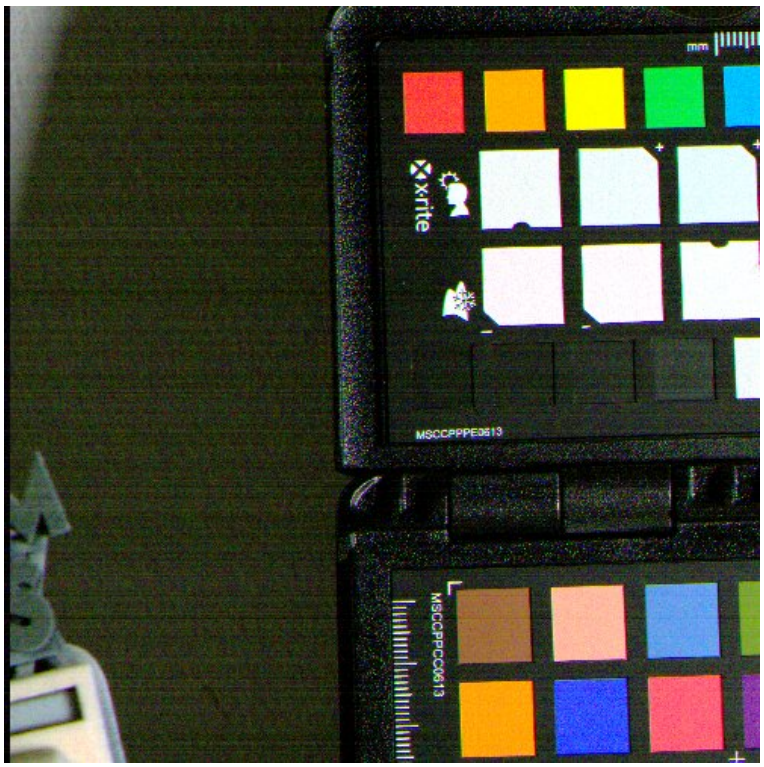
```
% 2-tap FIR, separate for odd/even rows
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --rnfilter=1 --check-darkfra
  Average     : 127.44
  Pixel noise : 2.84
  Row noise   : 0.51 (18.1%)
  Col noise   : 0.10 (3.4%)
```

```
% that ugly FIR that also looks at green channel differences
raw2dng blackframes-gainx1-offset2047-10ms-01.raw12 --swap-lines --rnfilter=2 --check-darkfra
  Average     : 127.47
  Pixel noise : 2.81
  Row noise   : 0.31 (11.1%)
  Col noise   : 0.10 (3.4%)
```

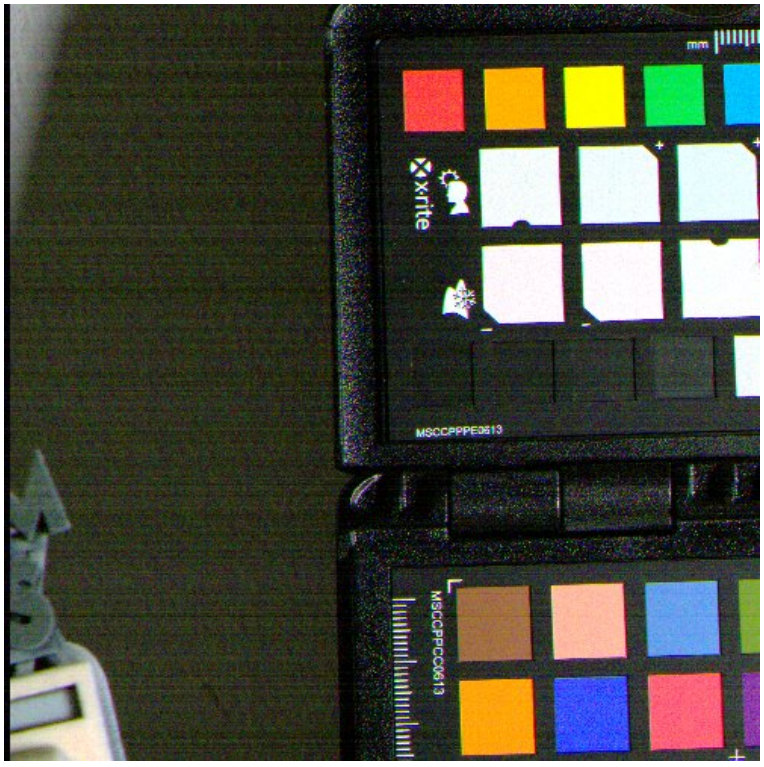
Example
Showing half-res image crops pushed by 4 stops (ufraw-batch –wb=auto –exposure=4 –shrink=2).

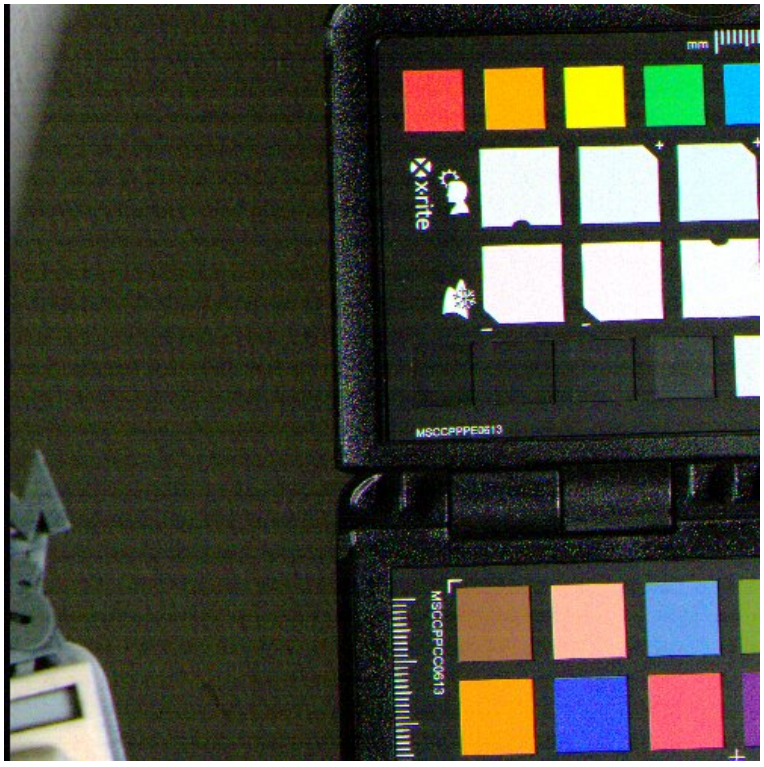
- Left: raw sensor data (adjusted black level manually).
- Right: after dark frame and dark current subtraction, but without correction from black columns.
- Note: in the raw data, even and odd rows have different black offsets; that's why we have wrong colors.



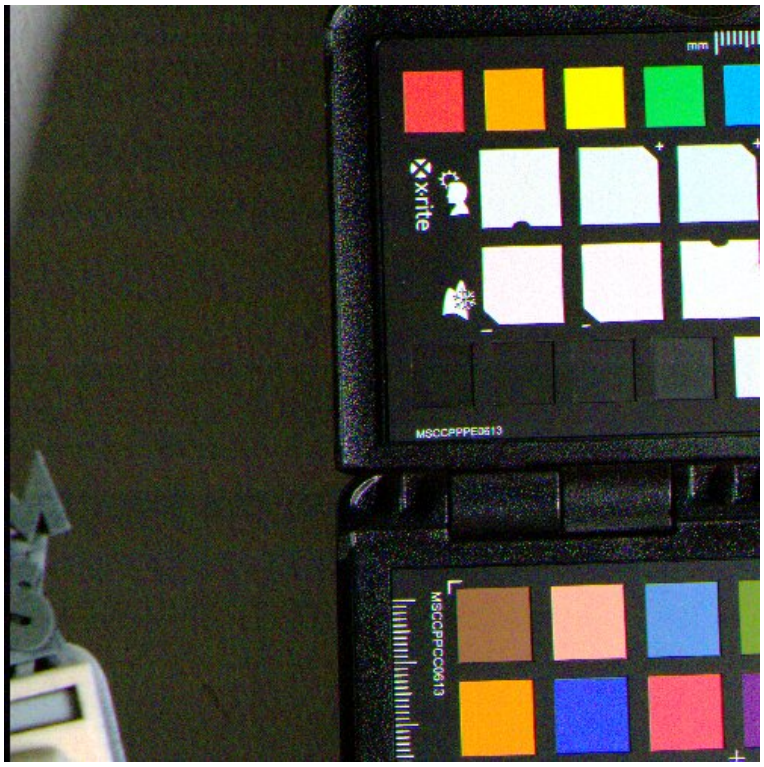


- Left: after dark frame, dark current and static offsets from black reference columns.
- Right: after dark frame, dark current and black reference columns correction, without removing the fixed frequency component.

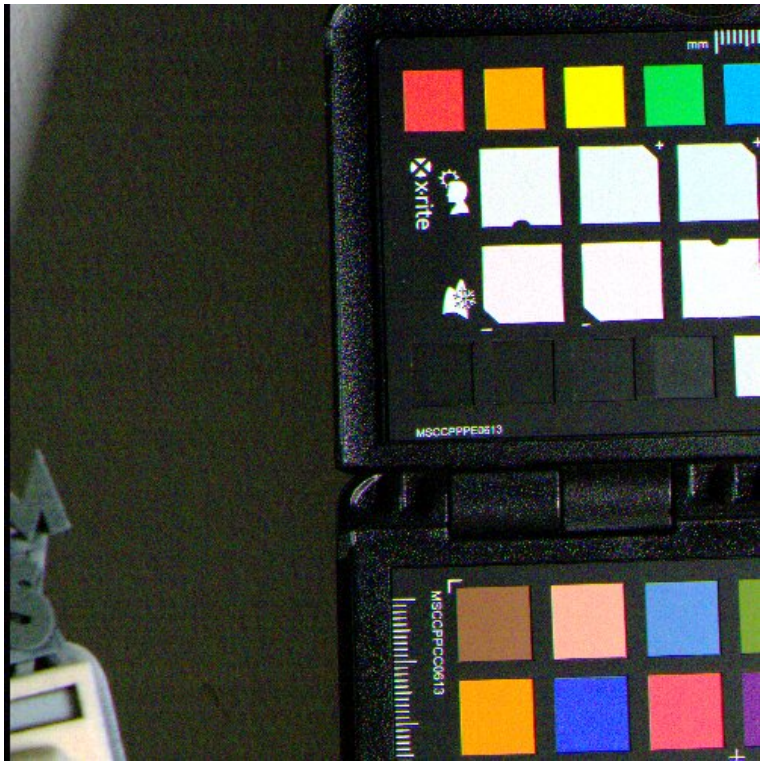




- Left: after dark frame, dark current and black reference columns correction.
- Right: after row noise reduction –fixrn



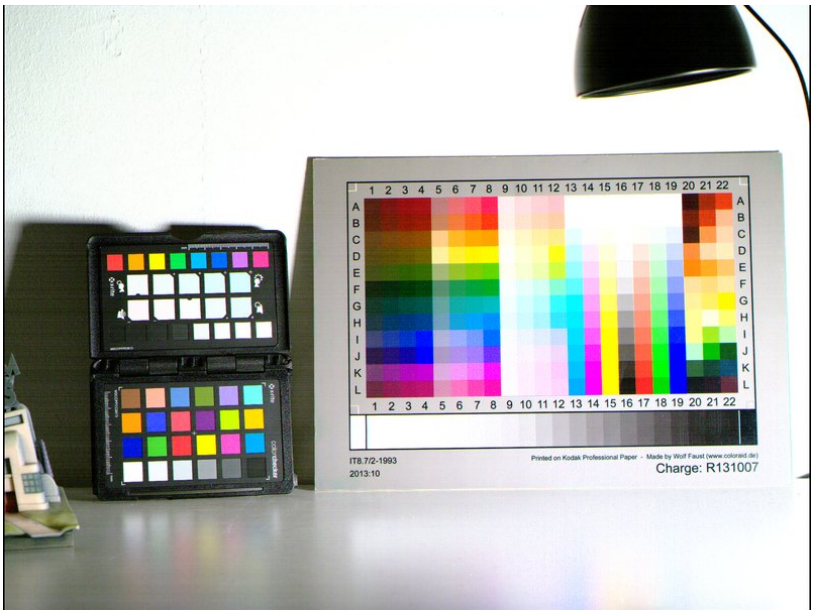
- Left: –rnfilter=1
- Right: –rnfilter=2



Algorithm internals for –fixrn
- Left: filtered image (vertical blur using a bilateral filter)
- Right: noise image, revealing row noise. Black regions are from edges that were masked.

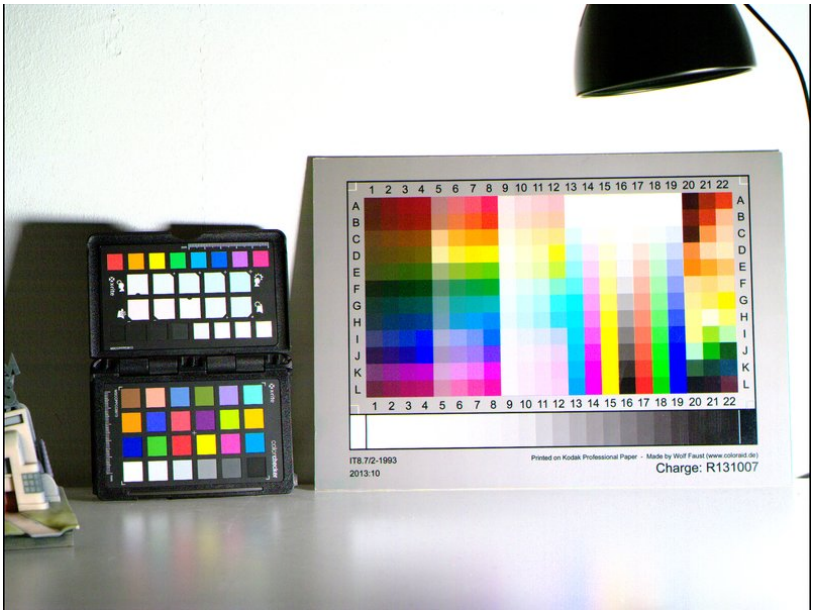
Downsized images

Corrected with dark frame and dark current only:

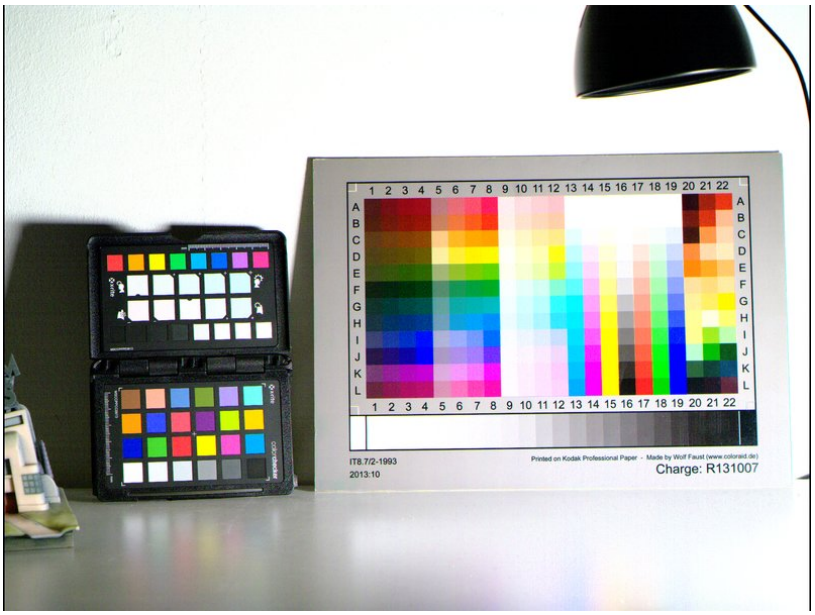


Also corrected with black columns:

Also corrected with –fixrn



Corrected with `–rnfilter=2`



Larger images (half-res):

- 10ms+4-no-blackcol.jpg - http://files.apertus.org/AXIOM-Beta/snapshots/pattern-noise/10ms+4-no-blackcol.jpg
- 10ms+4.jpg - http://files.apertus.org/AXIOM-Beta/snapshots/pattern-noise/10ms+4.jpg
- 10ms+4-fixrn.jpg - http://files.apertus.org/AXIOM-Beta/snapshots/pattern-noise/10ms+4-fixrn.jpg
- 10ms+4-rnfilter2.jpg - http://files.apertus.org/AXIOM-Beta/snapshots/pattern-noise/10ms+4-rnfilter2.jpg

10ms+4-no-blackcol.jpg
10ms+4.jpg
Final10ms+4-fixrn.jpg
10ms+4-rnfilter2.jpg

All files used for this test, including scripts, calibration frames and uncompressed images, can be found here: here.

image: 10ms.raw12
Calibration frames: darkframe-x1.pgm and dcnuframe-x1.pgm
Final image (–rnfilter=2): 10ms.DNG
Testing script: fpntest.sh
Script output: fpntest.log

11.3 Matrix Color Conversion

Each color channel is going through 4 stages that each look like this: (I+A)*E+O

```
I = input (25bit unsigned)
A = adjustment (30bit signed)
E = matrix coefficient (18bit signed)
O = offset (48bit signed)
```

The four stages are connected together that the offset of stage N is the output of stage N-1. The offset of stage 1 is set in registers (32-35). The output of stage 4 is the final result.

For each channel that results in the following fomular:

```
Ri = SUMj ((Ij + Aij) * Eij) +  Oi
```

Which in other words mean the result of matrix Eij multiplied with an input vector Ij + Aij plus an offset vector Oi .
Currently all registers (beside input/output) are shortened to 16bit signed (this could be changed easily) values so the matrix is currently limited to 1/256 of the possible accuracy (due to a 8bit shift of the output).

All input/output register are tested for over/underflow and clipped to 12 bit (this can also be disabled).

Setting/Getting Registers:

```
. ./hdmi.func # load  functions
```

```
mat_reg 0 # Read value of register 0 = A0 mat_reg 0 1 # Sett value of register 0 (A0) to
1
```

4x4 matrix coefficients (A0, A1, A2, A3, B0, B1, B2, B3, B4, C0, C1, C2, C3, D0, D1, D2, D3) are mapped to addresses 0 - 15. In registers 16 - 31 there are adjustment values and from 32 - 35 there are 4 offset values.

Input format per default is (R, G1, G2, B).

All values of the 4x4 matrix can be set conveniently with this script:

```
./mat4_conf.sh A0 A1 A2 A3 B0 B1 B2 B3 B4 C0 C1 C2 C3 D0 D1 D2 D3 O1 O2 O3
O4
```

Where the last quadruple is optional, it defaults to 0.

An RGB 3x3 matrix can easily be extended to a 4x4 matrix by using the coefficient for green and offsets twice or better to half both of the green coefficients and use the same value for both greens twice.

$$(I_R, I_G, I_G, I_B) \cdot \begin{pmatrix} M_{0,0} & M_{1,0} & M_{1,0} & M_{2,0} \\ M_{0,1}/2 & M_{1,1}/2 & M_{1,1}/2 & M_{2,1}/2 \\ M_{0,1}/2 & M_{1,1}/2 & M_{1,1}/2 & M_{2,1}/2 \\ M_{0,2} & M_{1,2} & M_{1,2} & M_{2,2} \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \\ X \end{pmatrix} = \begin{pmatrix} D2 \\ D1 \\ D0 \\ X \end{pmatrix}$$

11.3.1 mat4_conf.sh

To set the 4x4 color conversion matrix, you can use the mat4_conf.sh script:

The default configuration:

```
./mat4_conf.sh 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0  0
```

Note: In current AXIOM Beta firmware the mat4 order is changed and the default matrix is:

```
./mat4_conf.sh 0.3 0.3 0.3 0.3 0 0 0 1 0 0.42 0.42 0 1 0 0  0
```

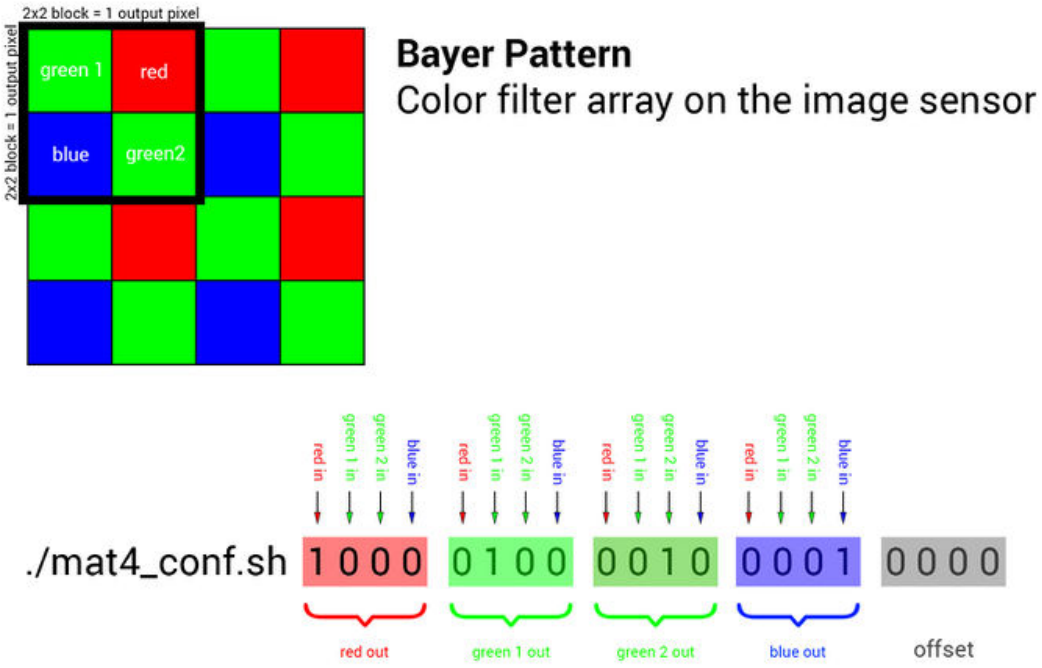
This will be changed to reflect the documentation again soon.

With the Blackmagic Video Assist (BMVA), a nicer matrix (ie better skin color, less greenish):

```
./mat4_conf.sh 0.3 0.3 0.3 0.3 0 0 0 1 0 0.3 0.3 0 1 0 0  0
```

This 4x4 conversion matrix is a very powerful tool and allows to do things like mixing color channels, reassigning channels, applying effects or doing white balancing.

TODO: order is blue, green, red!



4x4 Matrix Examples:

```
    ./mat4\_conf.sh  1 0 0 0  0 1 0 0  0 0 1 0  0 0 0 1    0 0 0 0   # unity matrix but not optim
    ./mat4\_conf.sh  1 0 0 0  0 0.5 0.5 0  0 0.5 0.5 0  0 0 0 1    0 0 0 0   # the two green chan
    ./mat4\_conf.sh  0 0 0 1  0 1 0 0  0 0 1 0  1 0 0 0    0 0 0 0   # red and blue are swapped
    ./mat4\_conf.sh  1 0 0 0  0 1 0 0  0 0 1 0  0 0 0 1    0.5 0 0 0    # red 50% brigther
    ./mat4\_conf.sh  1 0 0 0  0 1 0 0  0 0 1 0  0 0 0 1.5    0 0 0 0    # blue multiplied with fa
    ./mat4\_conf.sh  .25 .25 .25 .25  .25 .25 .25 .25  .25 .25 .25 .25  .25 .25 .25 .25
0 0 0 0    # black/white
    ./mat4\_conf.sh  -1 0 0 0  0 -0.5 -0.5 0  0 -0.5 -0.5 0  0 0 0 -1    1 1 1 1
# negative
```

11.4 CMV12000 PLR

Some attempts to reverse engineer the PLR high dynamic range mode from CMV12000.

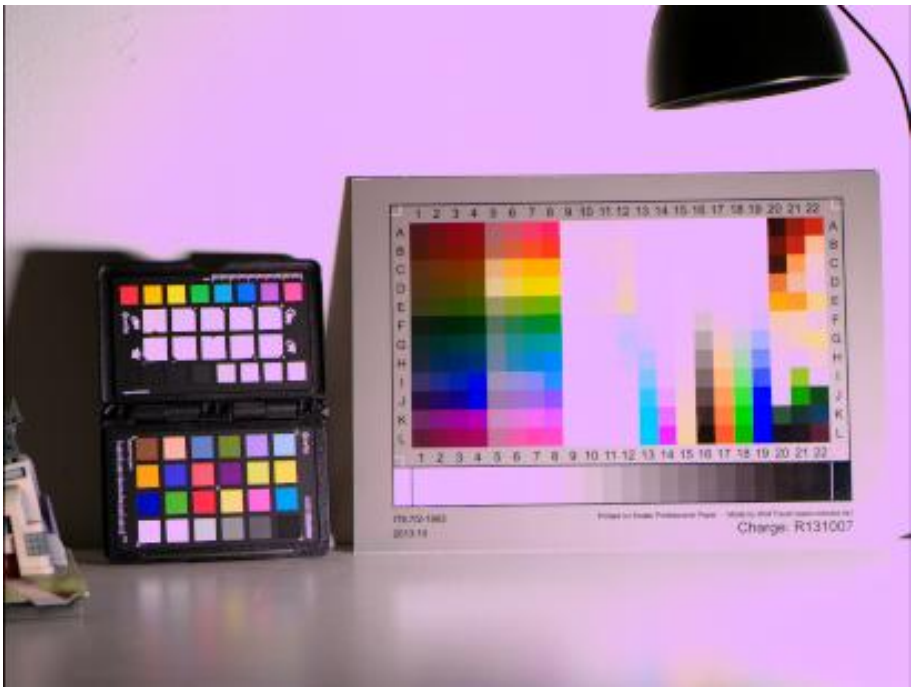
- 79: Number_slopes: 1,2,3.
- 75-78: Exp_kp1, Exp_kp2: exposure times for highlights (same formula as Exp_time)
- 106: Vtfl2, Vtfl3: knee point locations (range: 0-63; units: unknown)

Register effects.

I'll use an IT8 chart, exposed at 30 ms (normal exposure) and 100 ms (overexposed). A little dark in the lab today, but shouldn't be a big problem.

To analyze the images, I'll use octave 4.0, compiled with 16-bit image support. The scripts should run in Matlab as well, with minimal changes.

11.4.1 Linear exposures

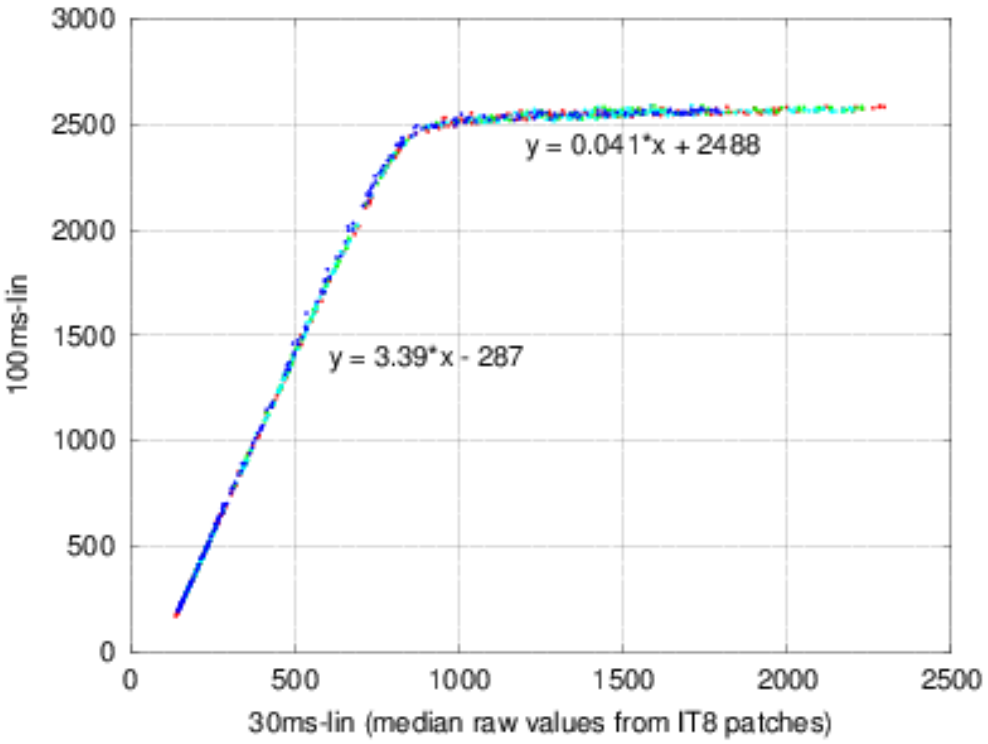


Let's check if the first image is really exposed to the right, in octave.

```
a = read\_raw('30ms-lin.DNG');
prctile(a(:),99) - 128        % note: black level is forced to 128 in raw2dng
ans =  2269                   % clipping starts at about 2400-2500 above black
```

Let's check if the clipping point is, indeed, where I say:

```
figure, hold on
colors = 'rgcb';                                % meaning: red, gren1, green2 (plotted as c
[g30,c30]   = sample_it8('30ms-lin.DNG', 0);    % read median RGGB swatch values from IT8 c
[g100,c100] = sample_it8('100ms-lin.DNG', 0);   % first output arg is grayscale, second is
for i = 1:4
  plot(c30(:,i), c100(:,i), ['.' colors(i)]);   % plot the 100ms image vs the 30ms one (ext
end
```



- This approximates (but it's not equal to!) the response curve without PLR.
- This plot assumes the sensor response in the 30ms image (which was not overexposed) is linear (but it's probably not).
- The sensor doesn't clip very harshly to white (could be the PLR circuits kicking in with a very low exposure time? to be checked).

11.4.2 2-segment PLR exposures

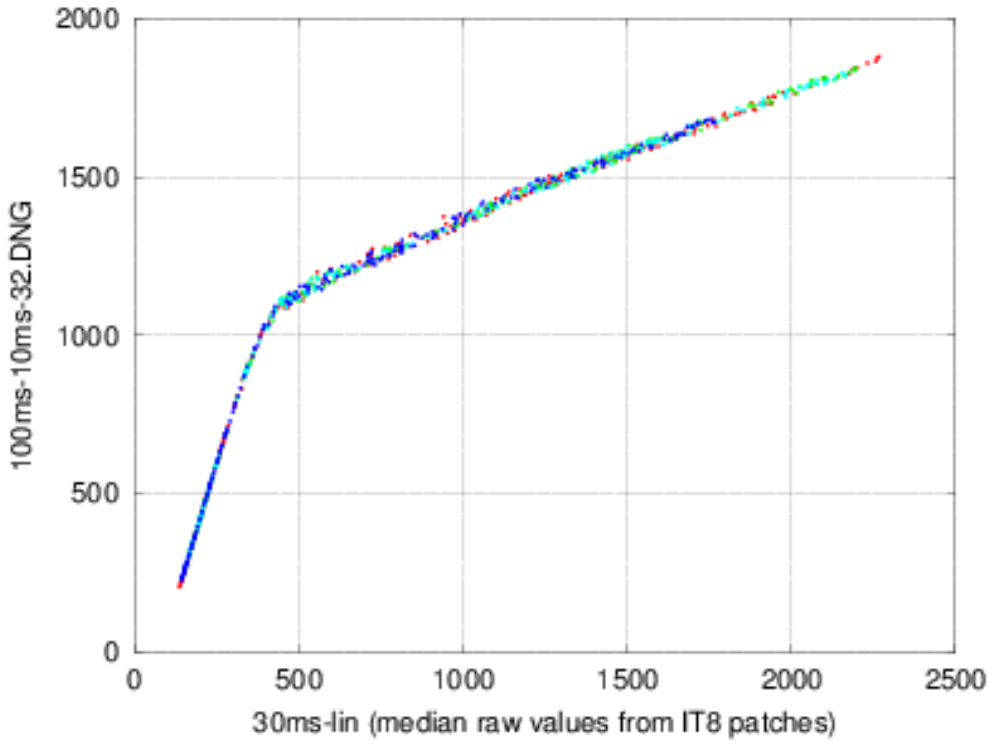
Let's start with a 2-segment PLR exposure, 100ms/10ms, vtfl2=32. That means, Number_slopes = 2, Exp_time = 8072, Exp_kp1 = 805, Vtfl=96.

Although the image may look a little overexposed at first sight (because of the magenta cast), the raw levels seem to be alright:

```
c = read\_raw('100ms-10ms-32.DNG');
prctile(c(:),99)
ans =  1957
```

Let's check our response curve approximation:

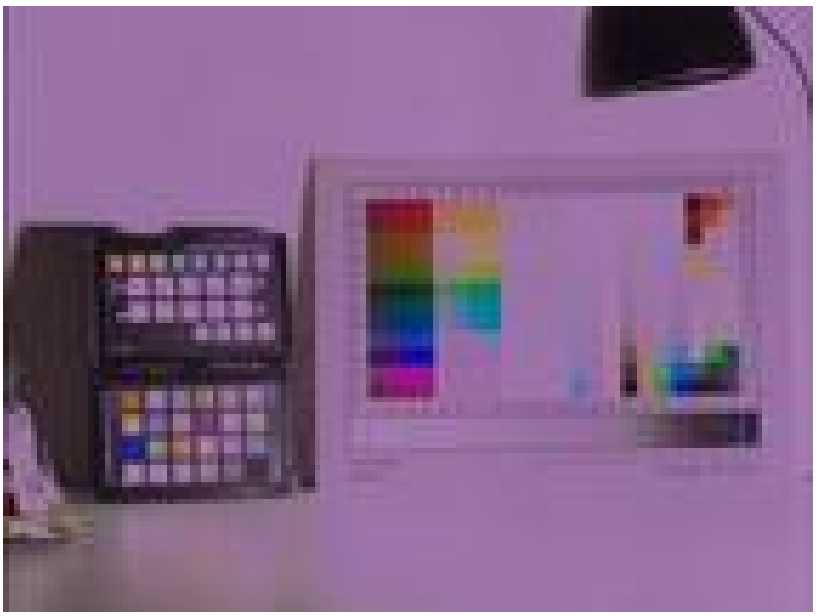
```
[g100p,c100p] = sample\_it8('100ms-10ms-32.DNG', 0);
figure, hold on
for i = 1:4
  plot(c30(:,i), c100p(:,i), ['.' colors(i)]);
end
```

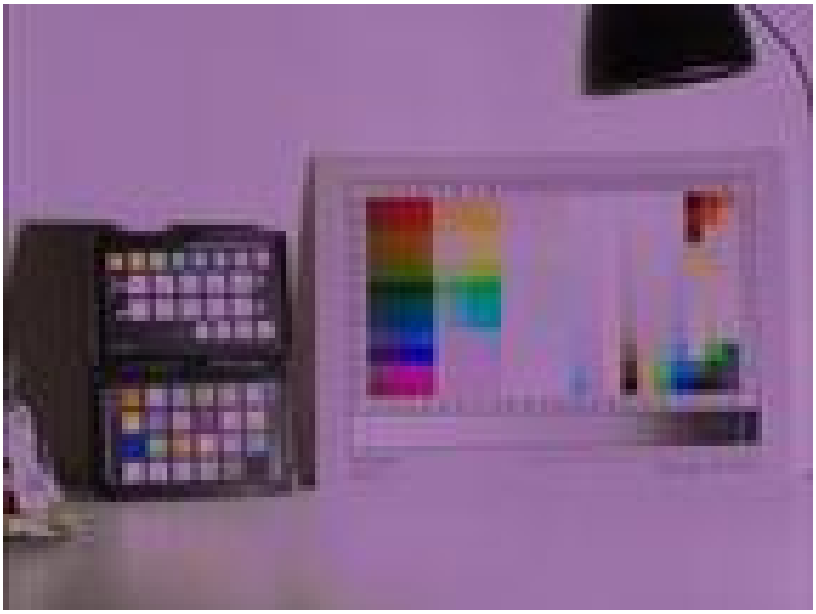


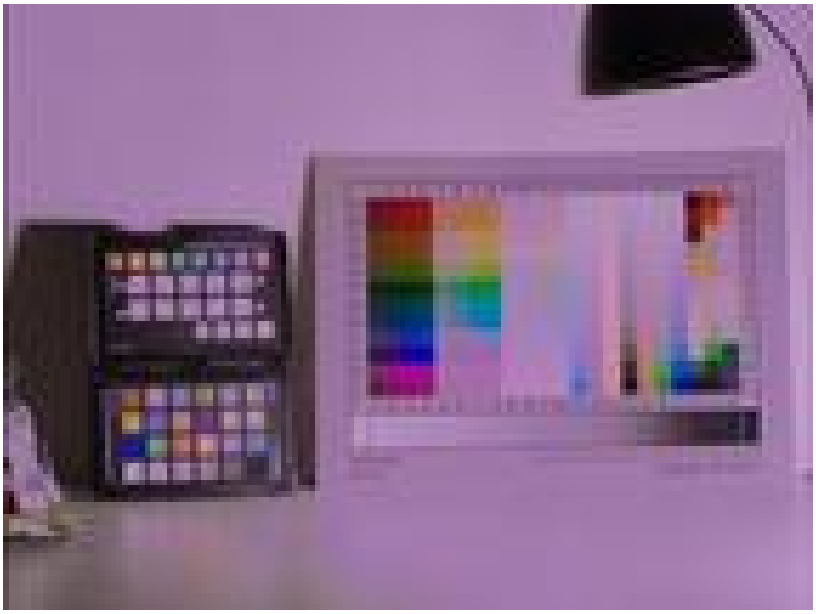
- The knee point isn't as sharp as in the datasheet.
- Its location is around 400 (todo: find the relationship between this and vtfl).
- The second exposure segment appears a little noisy .

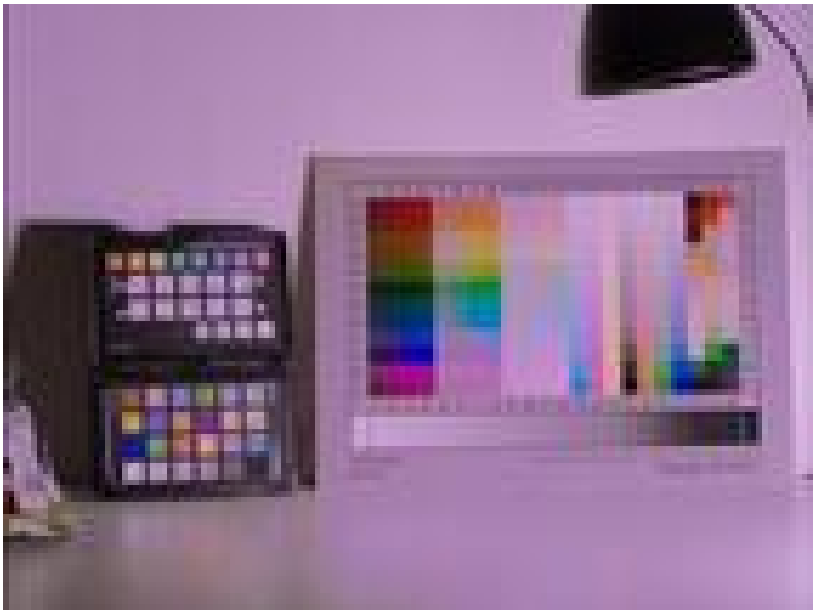
11.4.3 Changing Exp_kp1

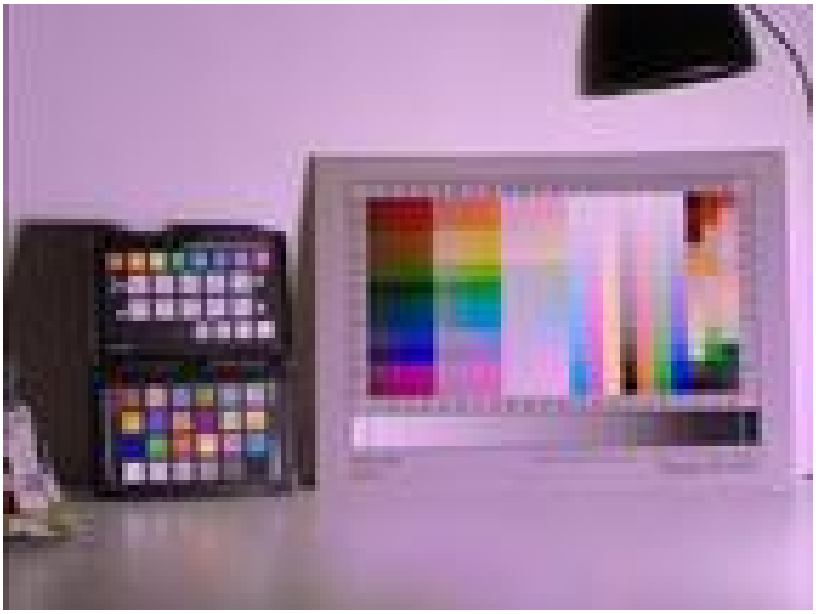
Let's leave Vtfl constant and change exposure time for the second segment (0,1,5,10,20,35,50 ms).



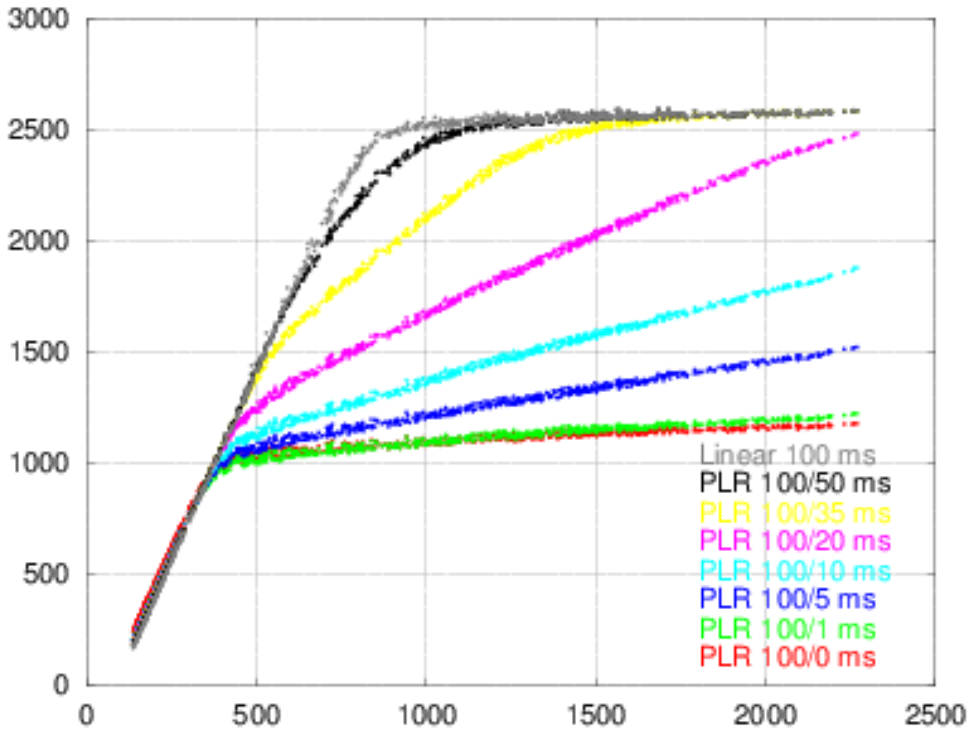






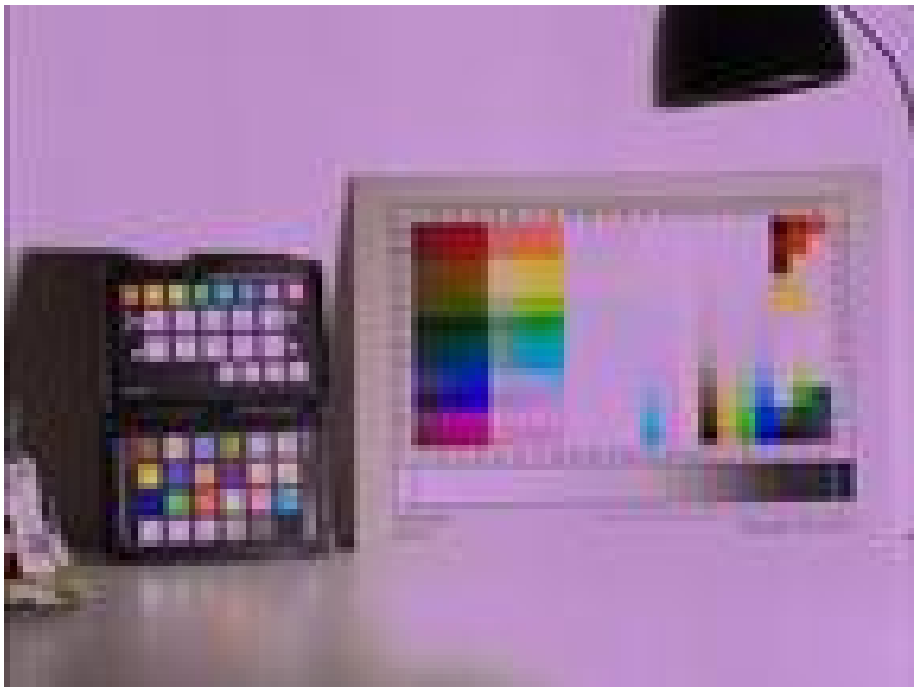


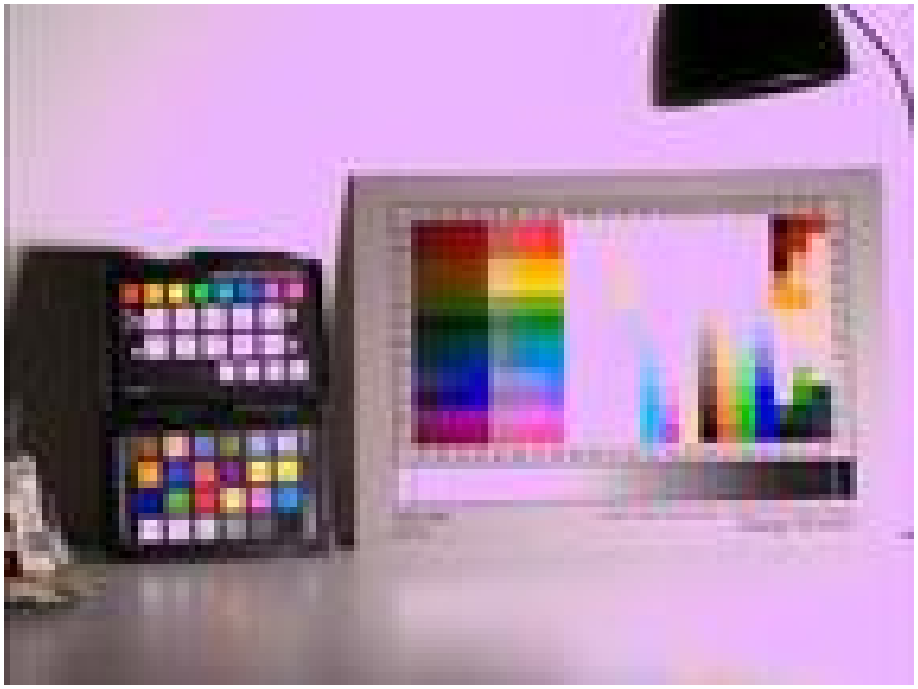
Commands for plotting the "response curve" approximations are left as an exercise to the reader. Result:

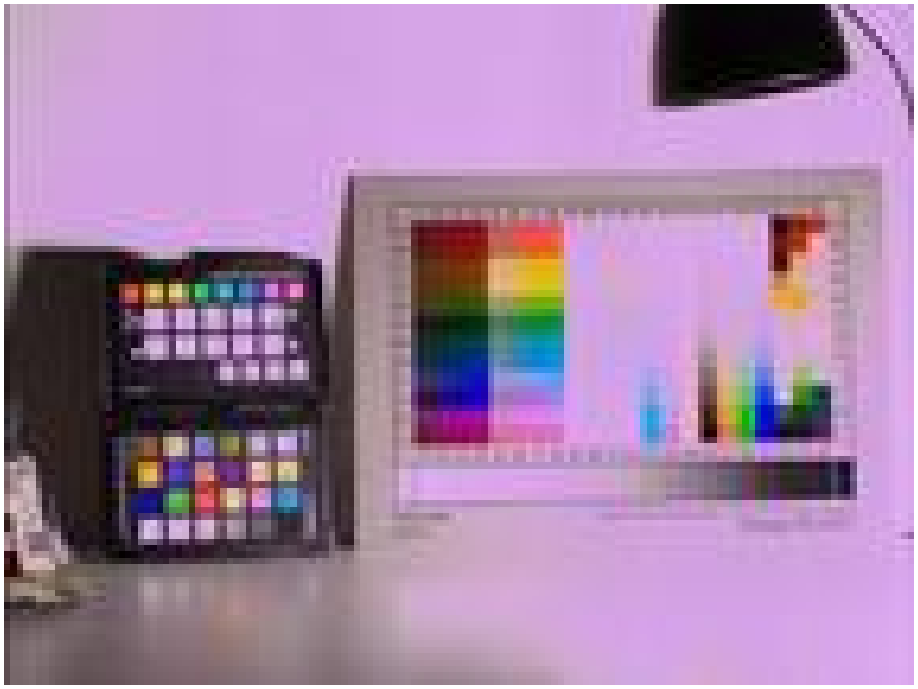


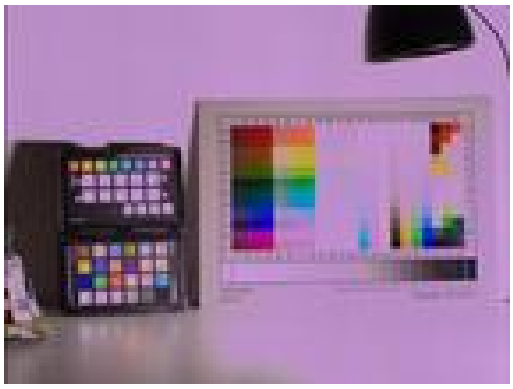
11.4.4 Changing Vtfl

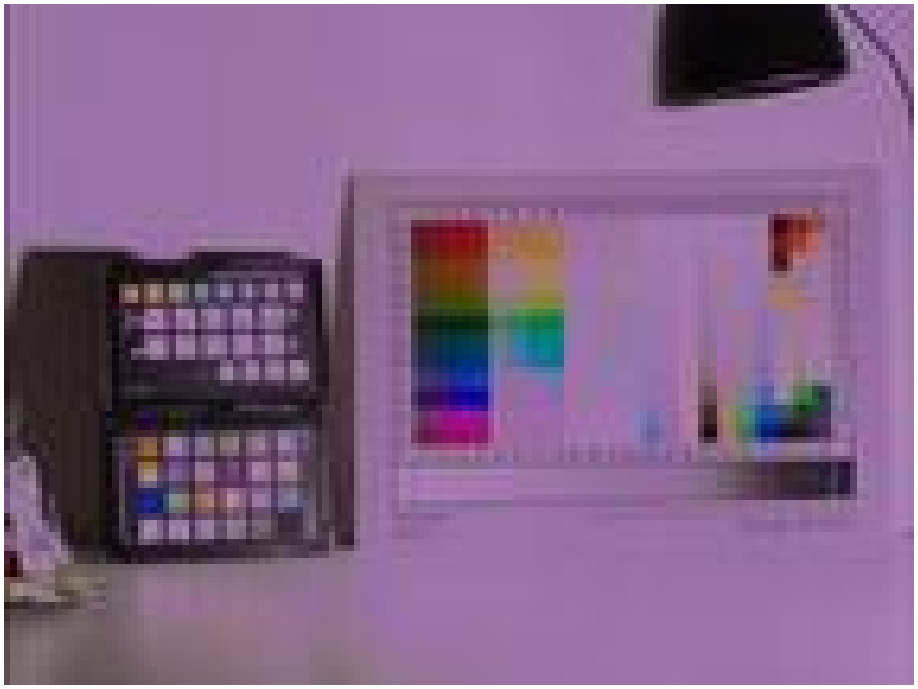
Now let's leave exposure time constant (100ms/1ms) and change Vtfl (off,0,8,16,24,32,40,48,56,63):

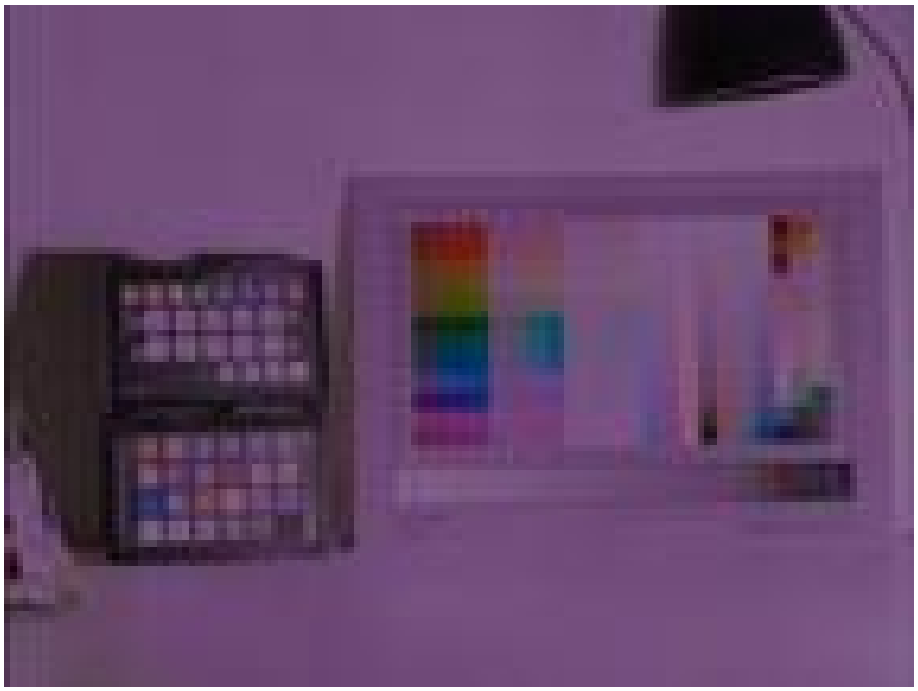


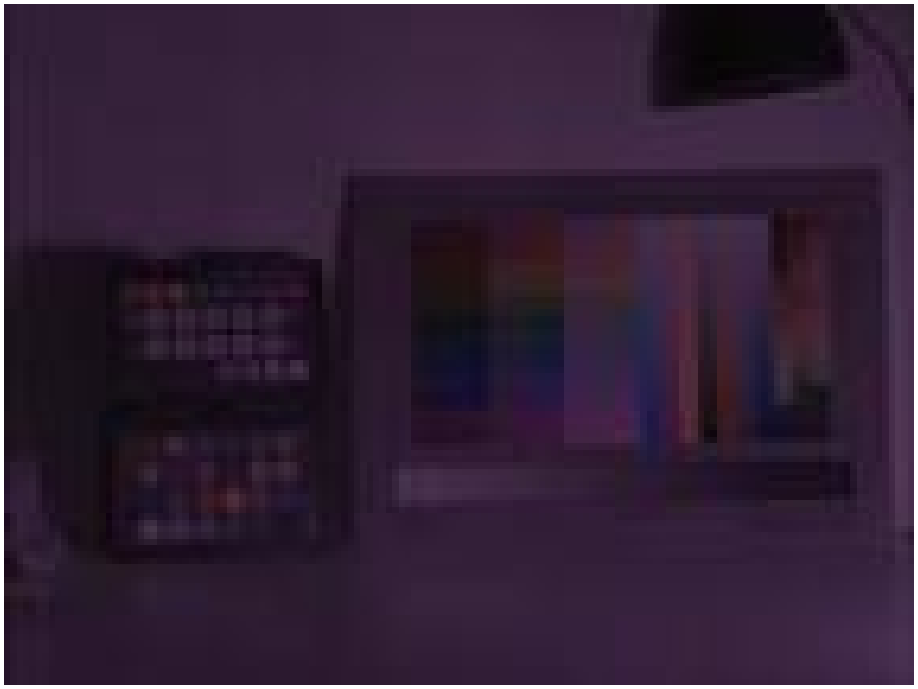


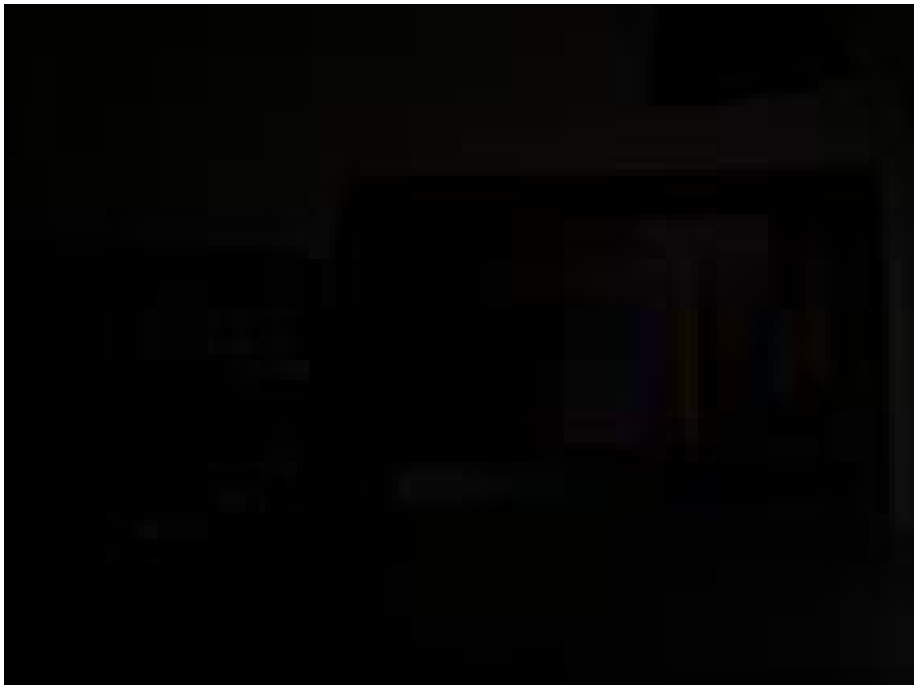


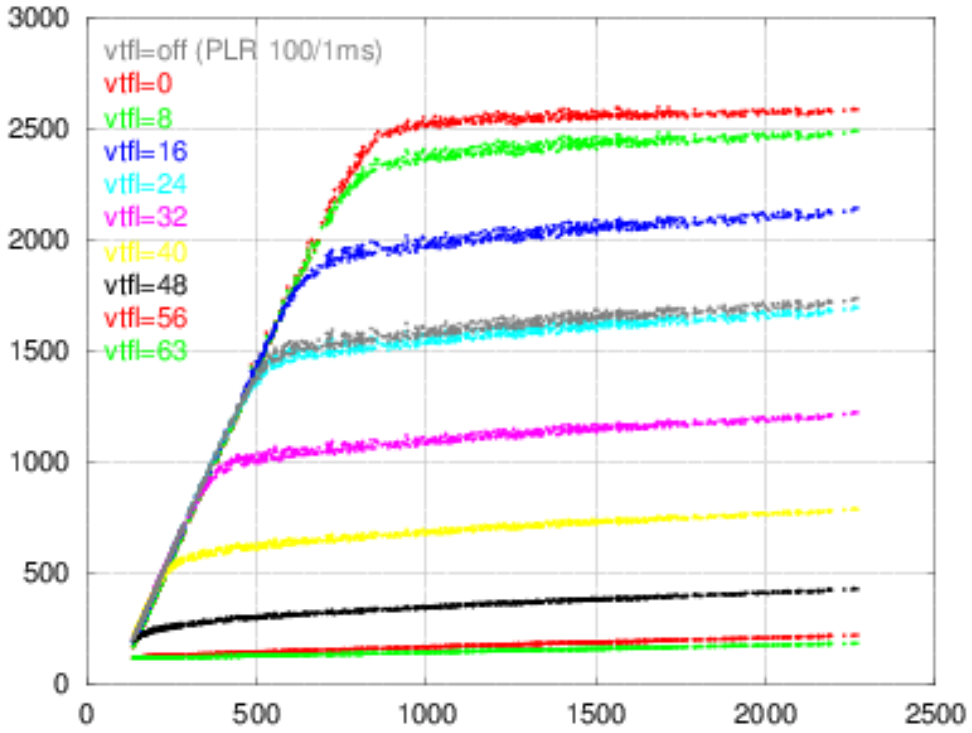






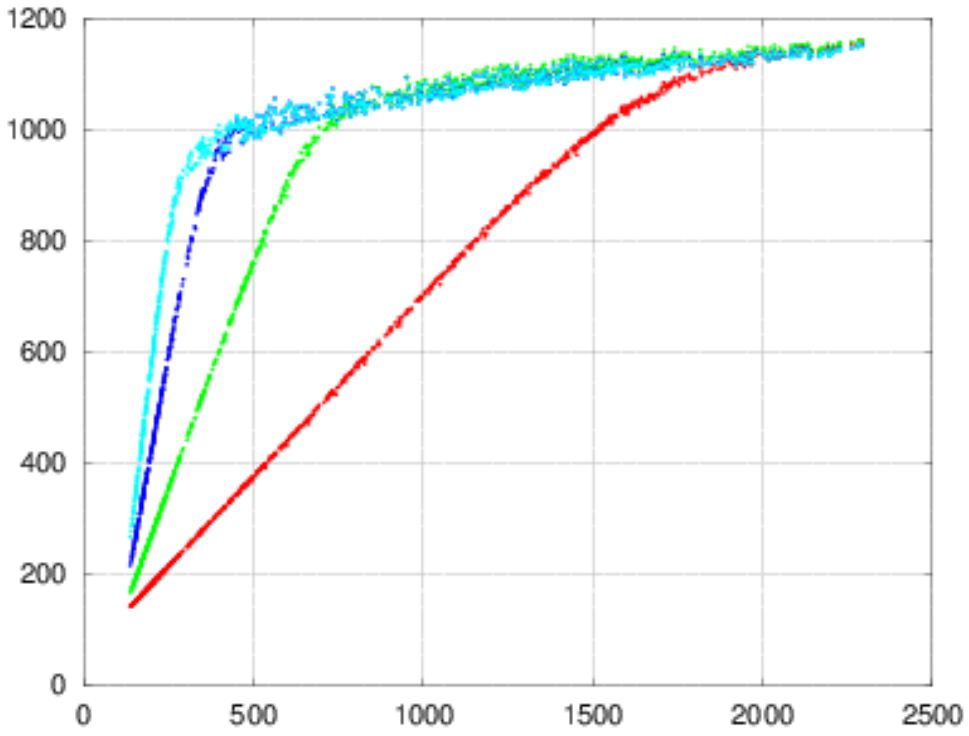






11.4.5 Changing Exp_time

Now let's leave Exp_kp1 and Vtfl constant (0ms/32) and change base exposure time (20,50,100,150 ms):



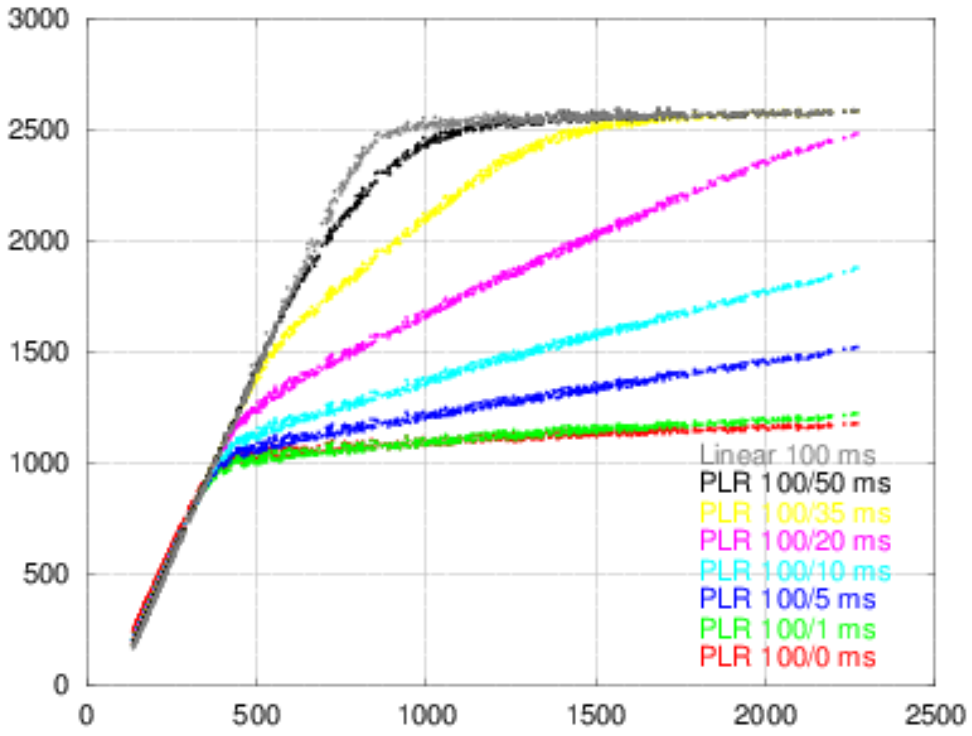
- There is a noticeable light leak in the second segment, but it doesn't seem to change with base exposure time.
- Most of the noise in these curves seems static (correctable with calibration frames).

11.4.6 Changing Exp_kp2 and Vtfl3

As expected, these settings do not seem to have any noticeable effect when num_slopes = 2.

11.4.7 Mathematical models

First models of the response curve for 2-segment PLR exposures.
Let's go back to the scenario with constant Vtfl and variable exposure.

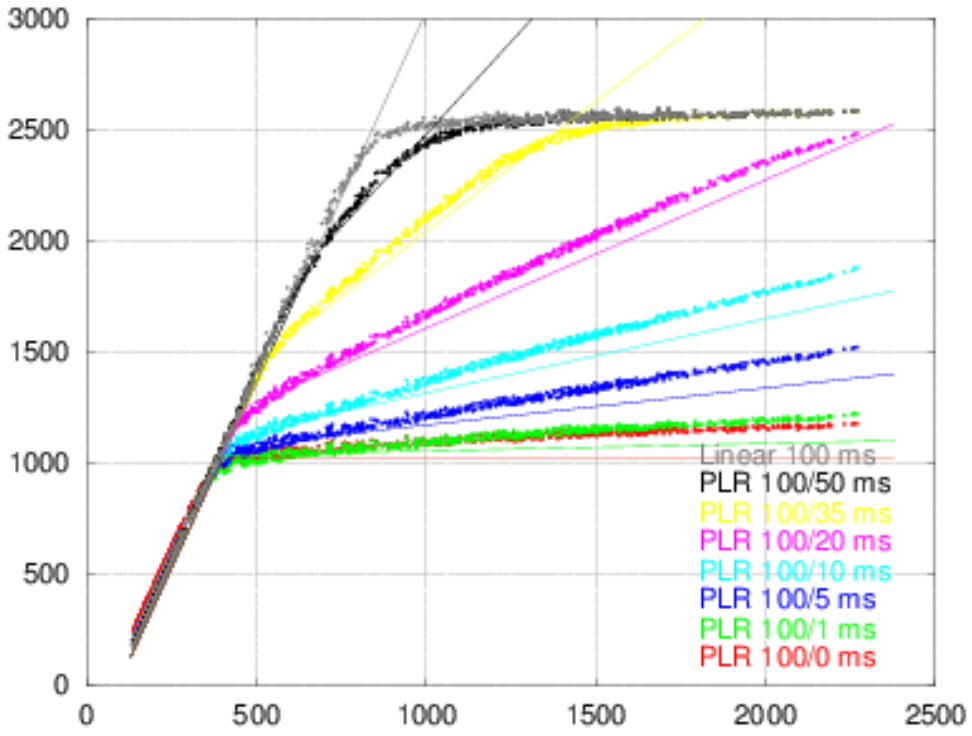


Notice the knee point appears to move to the right with higher Exp_kp1 values. Why would this happen?

If we assume the knee point threshold (Vtfl2) is only monitored while we still have time to fully expose the second segment (Exp_kp1), we can imagine a model of the PLR exposure that looks like this:

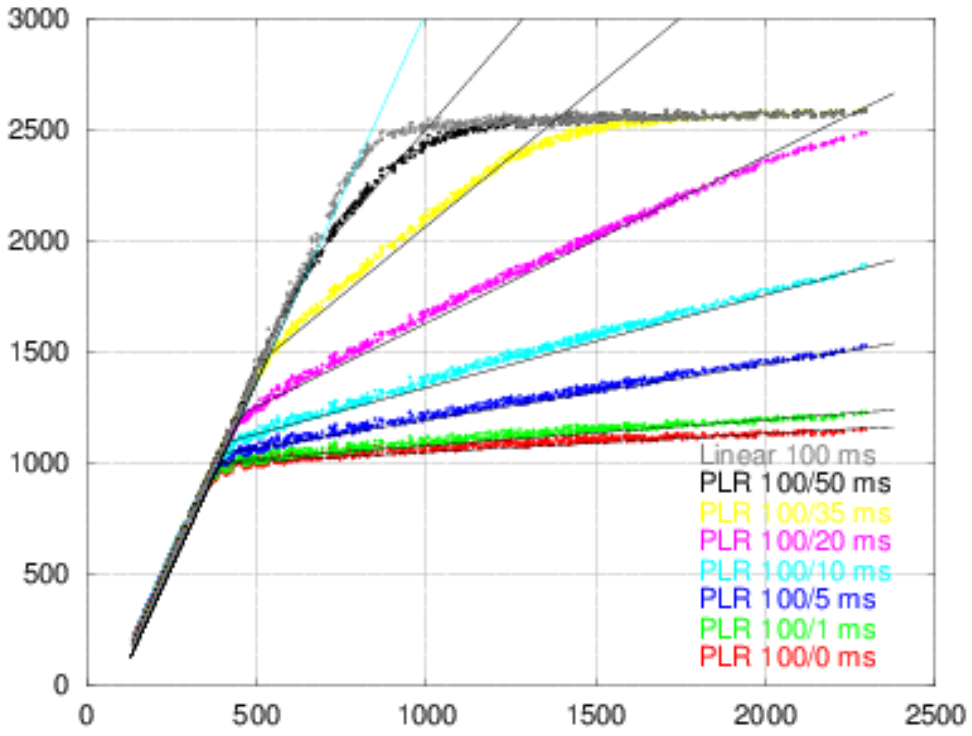
```
function y = plr\_model\_0(x, exp\_time, vtfl\_thr, exp\_kp1)
   % regular exposure time
   y = x .* ex\p_time;

   % time needed to reach the Vtfl threshold (lower input levels => longer times)
   t\_reach\_vtfl = vtfl\_thr ./ x;

   % which pixels reach vtfl?
   % assume this threshold is only monitored while we still have time
   % to fully perform the exp\_kp1 (that is, before exp_time - exp\_kp1)
   reached\_vtfl = t\_reach\_vtfl < exp\_time - exp\_kp1;

   % how the exposure behaves in the highlights
   y\_highlight = vtfl\_thr + x .* exp\_kp1;

   % copy highlight values only for those pixels that reached Vtfl
   y(reached\_vtfl) = y\_highlight(reached\_vtfl);
end
```

Let's see how it compares to our real data.



Notice the slopes of the second segment seem to be higher in the real data, compared to our model, as if the real exposure time would be a little higher.
Since the slope at `exp_kp1=0` doesn't depend on total exposure time, let's assume there is some sort of exposure leak - that is, the actual exposure time is `exp_kp1` + some constant (called `exp_leak`).

```
function y = plr\_model\_1(x, exp\_time, vtfl\_thr, exp\_kp1, exp\_leak)

   [ ... ]

   % how the exposure behaves in the highlights
   y\_highlight = vtfl\_thr + x .* (exp\_kp1 + exp\_leak);

   [ ... ]

end
```



Tuning parameters (trial and error): vtfl_thr = 850, exp_leak = 2.5ms.

Black level 128 (after subtracting black reference columns, dark frame and dark current nonuniformity). Reference frames obtained from fitting 256 dark frames taken at exposures from 1 to 64 ms, in 1 ms increments, 4 images at each exposure, with raw2dng –swap-lines –calc-dcnuframe *x1*.raw12.

This model appears to explain the actual response curves pretty well, but there's still room for improvement.

At this point, I would try to find some real response curves and reduce the noise of the test images by averaging multiple frames, then repeat the experiment. This will take a while.

After averaging 50 frames at each setting, the noise in the curve remained the same, so it must be systematic error (correctable with some sort of calibration frame).

Finding response curves from bracketed images is difficult - we can't just scale exposure times hoping we'll get the same curve. We will need a dimmable light source with no ambient light (or some dedicated calibration hardware).

To be continued.

11.5 CMV12000 Response Curves

Identifying response curves on the CMV12000 sensor

We would like the data coming from the sensor to be linear (that is, proportional to the number of photons received). Plus some constant offset.

Is it already linear, or do we have to adjust it?

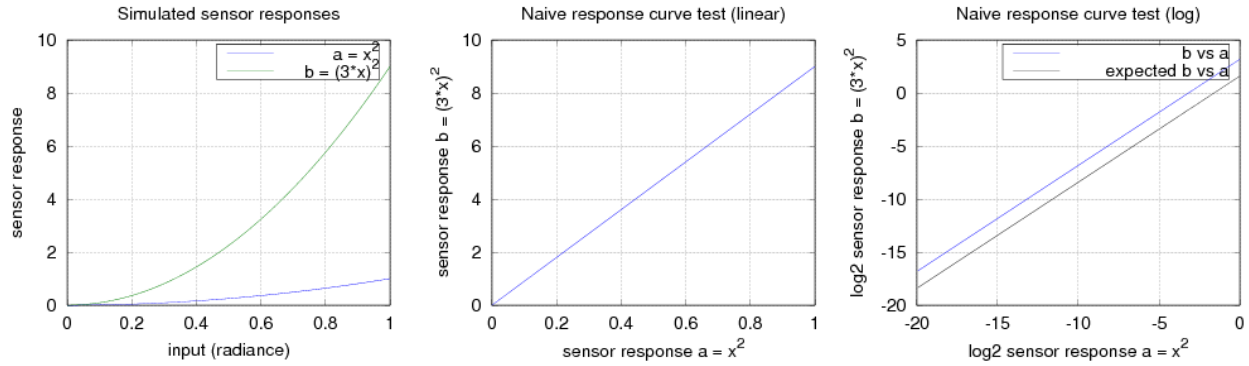
This sensor has a high dynamic range mode called PLR (piecewise linear response). In this mode, the sensor response is highly nonlinear (configurable, but the output is not exactly piecewise linear, so we still have to identify the curves). Details about this mode on the PLR page.

Here we'll focus on identifying the response curves.

eg: input range where the linear sensor response, and use that range to correct nonlinearities that occur outside this range (for example, with underexposure, overexposure, or by comparing a PLR mode with a regular exposure mode). This can be good enough for initial tests, but it has a major problem.

Suppose we have a sensor with this response: $y = (t*x)\hat{2}$ (where t is exposure time, x is radiance, x*t is the number of photons captured, and y is the sensor output value, all in arbitrary units). Let's expose two synthetic images from it, in octave:

```
x = linspace(0, 1, 1000);              % synthetic "image" with range from 0 to 1 (black to wh
a = x.^2;                              % "expose" the image for 1 time unit
b = (3*x).^2;                          % "expose" the image for 3 time units
subplot(131), plot(x, a, x, b)         % plot the sensor responses vs input signal (radiance)
subplot(132), plot(a, b);              % plot the second image, under the assumption that firs
subplot(133), plot(log2(a), log2(b)); % also try a logarithmic plot
hold on, plot(log2(a),log2(3*a),'k'); % expected response for the logarithmic plot
```



Our simulated sensor has obviously nonlinear sensor response, yet comparing two images suggest its output might be actually linear.

However, the log plot indicates there may be a problem, so this type of nonlinearity is not really 100
This nonlinearity was pretty obvious, but subtler variations may be much more difficult to spot, so we should really find a way to recover the true response curve. Without a photon counter, that is.

Test data

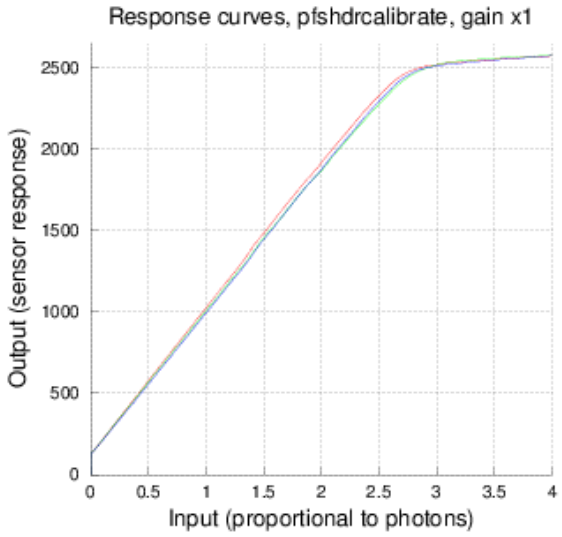
Bracketed image of the grayscale line from an IT8 chart, 1...100ms in 1ms increments, exposure sequence executed 100 times, images at identical settings averaged.

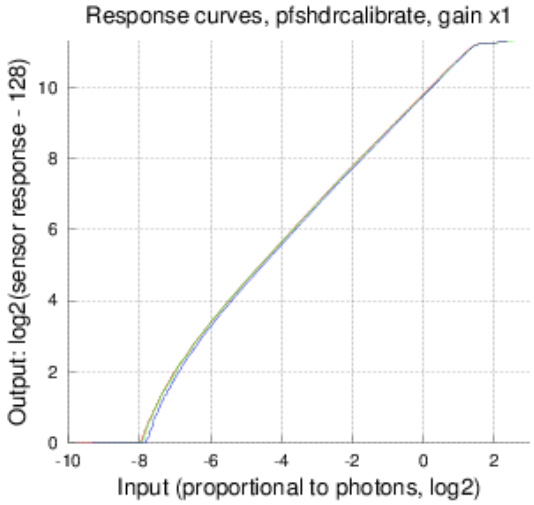
Existing algorithms

- Review: Best algorithms for HDR image generation. A study of performance bounds. Ref.
- Debevec97 Ref 01. kRef 02, implemented in mkhdr Ref 03.
- Robertson02 Ref 01, implemented in pfshdrcalibrate Ref 02.
- ArgyllCMS, shaper+matrix algorithm
- other algorithms that we can try?

11.5.1 Debevec97 results

Robertson02 results





These curves seem to indicate that our black level may be a little too high. Figure out why.

Scripts, logs: http://files.apertus.org/AXIOM-Beta/snapshots/response-curves/pfs/

11.5.1.1 ArgyllCMS shaper+matrix results

WIP, source code - https://github.com/apertus-open-source-cinema/misc-tools-utilities/commit/155be5e8cac9c7d158b8cf1a3055c833c9eab9a9.

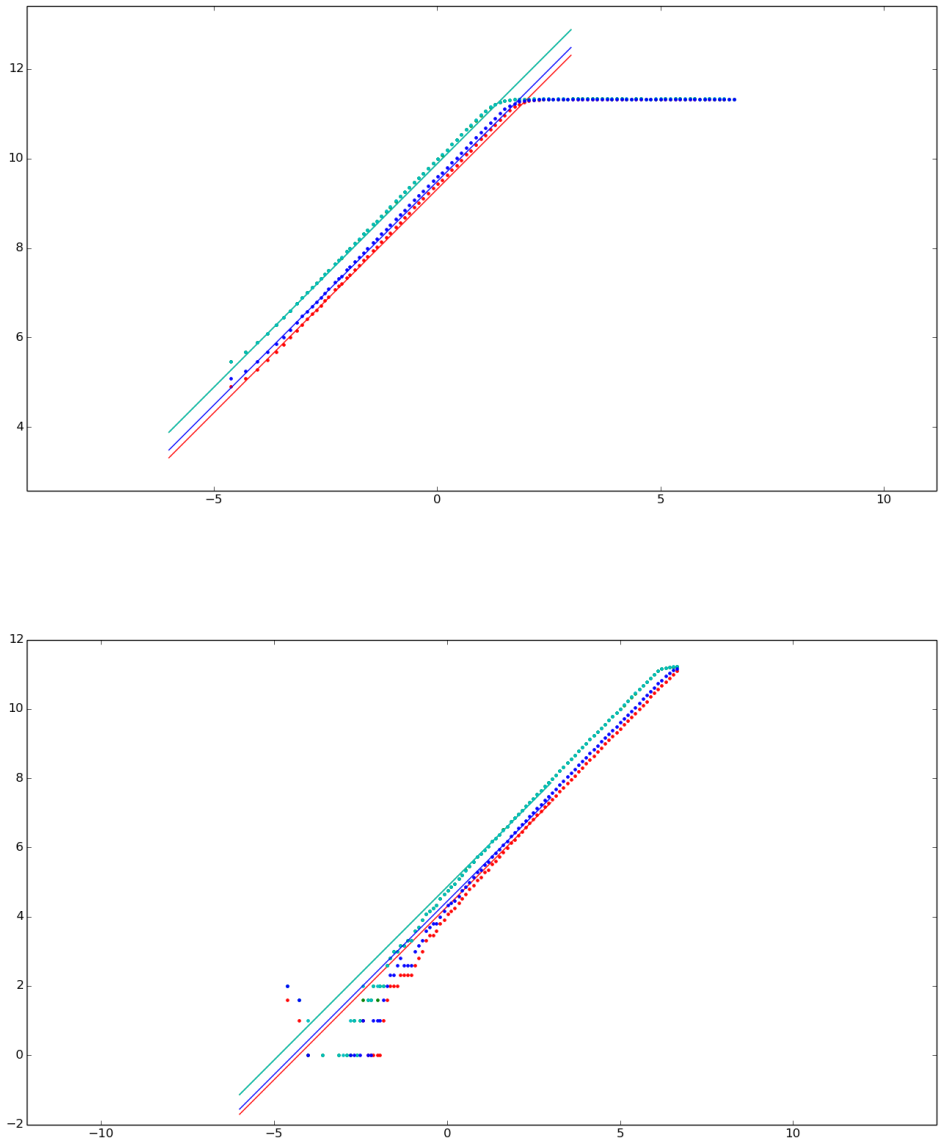
Q: can this be used on bracketed images?

11.5.2 Custom algorithms

11.5.2.1 Median vs exposure

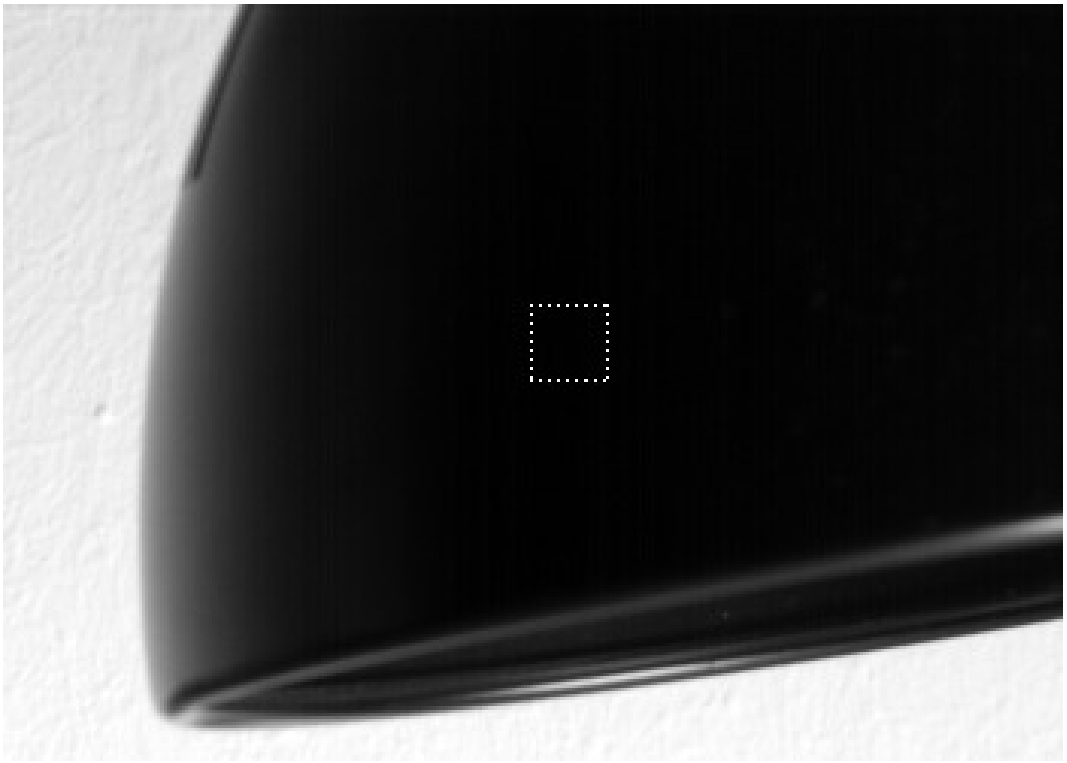
This assumes the exposure setting from the sensor is accurate, so the input signal that will be digitzed is proportional to number of photons (scene radiance multiplied by exposure time.

Unfortunately, this method delivered inconsistent results (two experiments resulted in two different curves -



11.5.3 Black Hole anomaly

Here's an example showing how much the exposure setting can be trusted. Crop taken from a bracketed exposure (1,2,5,10,20,30 ms) repeated 700 times and averaged. Image from the figure is from the 30ms exposure.



That's right - when exposure time increases, very dark pixels become even darker.

Also note the black level, after correcting the image with dark frames and black reference columns, is set to 128 (the entire image is adjusted by adding a constant, so the black reference columns reach this value). The pixels showing this anomaly are below this black level. There is detail in the image, even in a single (non-averaged) exposure, but for some unknown reason, it ends up below the black level.

This behavior cannot be reproduced on dark frames though, so probably (unconfirmed hypothesis) the black level goes down (not sure if globally or locally) when the image content gets brighter. The detail in the dark area on the test image looks normal (not reversed), so probably the total number of photons captured is the variable we should account for?

11.5.4 Matching per-pixel curves

This algorithm is roughly inspired from Robertson02, but without any solid mathematical background; only with the hope that it will converge to a good solution.

1. Initial guess:

- Plot per-pixel response curves on a graph, trusting exposure control on the sensor for accuracy.
- For each sensor output value, let's say from 16 to 2048 in 0.25-stop increments, compute the median exposure required to get that output.
- Shift each curve horizontally, in log space, to match the median exposure.
- Repeat until convergence.

1. Refinement:

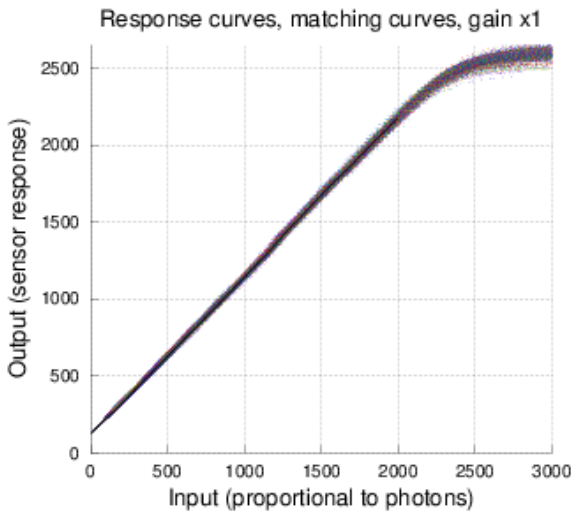
- Assume each pixel curve may be shifted by a constant offset vertically.
- Repeat the same algorithm used for initial guess, but also shift the curves vertically, in linear space, by a constant offset.
- Assume the average shift value is 0 (the algorithm may converge to a wrong solution without this constraint).

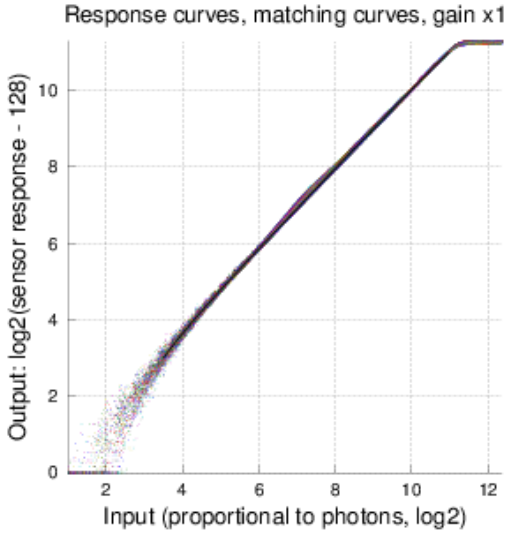
Problems:

- You can't operate on too many pixel curves at once (it can get slow and memory-intensive).
- There's no proof on whether it will converge, and if yes, how accurate the solution will be (but we can try it on synthetic data).

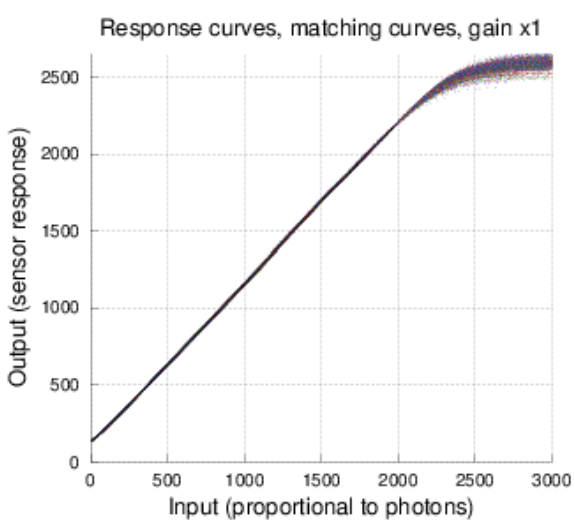
Example (only a single line from the test images was used):

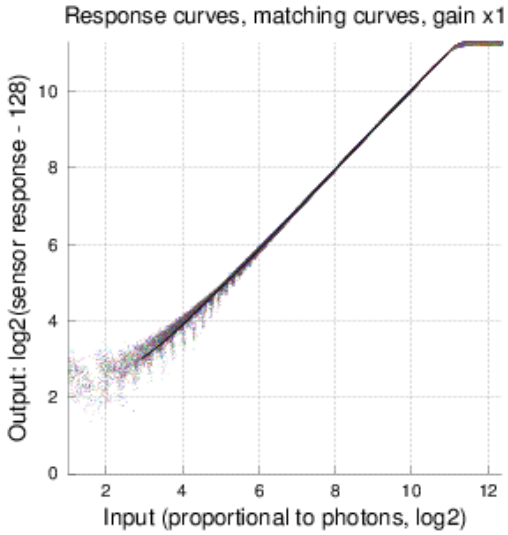
Initial guess:





Second guess:





Note: The response looks a bit different from Robertson02. Who is right?

11.5.5 Direct per-pixel curves

Storing response curves for each pixel would be really expensive in terms of storage (memory), but may be worth trying.

Test data: bracketed exposures at 1,2,5,10,20,30 ms. 700-frame average for each exposure (total 4200 images, captured overnight). This would reduce dynamic noise by log2(sqrt(700)) = 4.7 stops, so the static variations should be reasonably clean.

Since the test image was not perfectly uniform (since we didn't actually shoot a flat field frame), we probably won't be able to find out the PRNU.

11.5.6 WIP

11.5.6.1 Curves from grayscale IT8 reference data

Response curves can be also estimated from the IT8 reference data, which was (hopefully) measured with much better accuracy than what we can achieve with our current setup.

Advantage:

- We have some absolute reference data.
- We can get a quick estimation of some part of the curve from a single image.

Problems:

- In our setup, the illumination is nonuniform.
- Tthe dynamic range of the IT8 chart is not very high (7.33 stops on the bottom gray scale).
- Few data points (because, without a matrix, we can only use grayscale swatches; on the good side, these

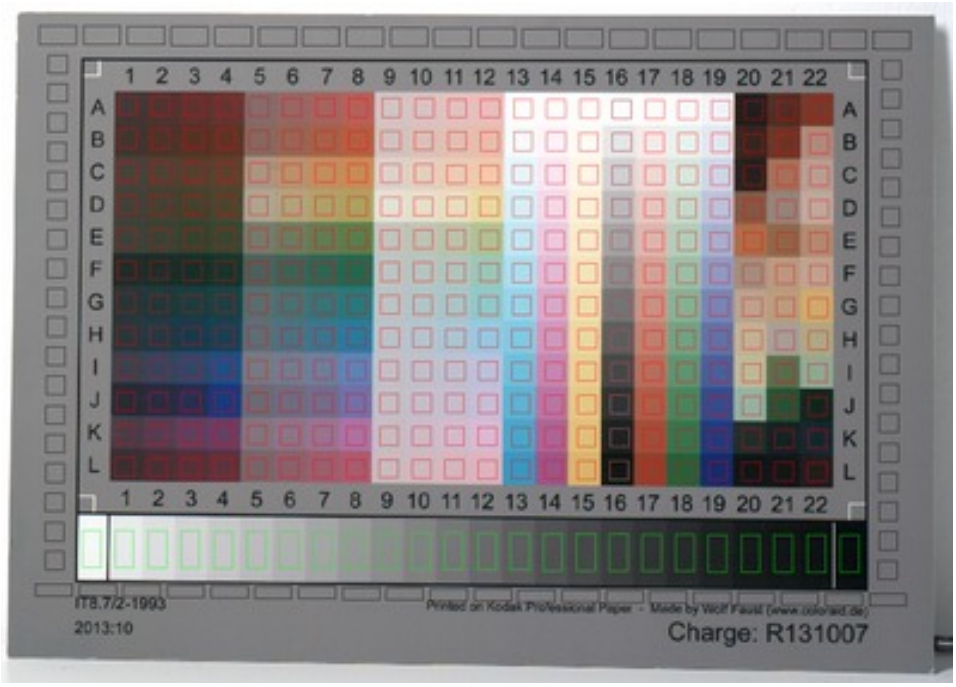
few data points are not very noisy).

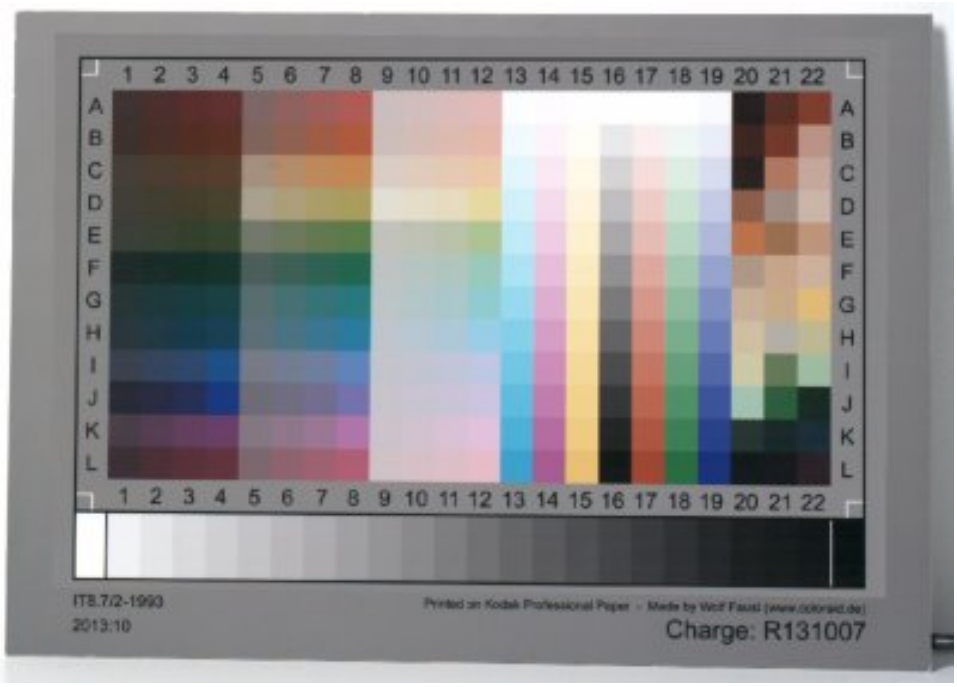
Solutions/workarounds:

- We can account for the nonuniform illumination by checking the brightness levels at the edges of the chart (light gray); it's not very exact, just better than nothing.
- To cover the entire dynamic range, we can use bracketed images, but this is not perfect - the image levels appear to vary with number of photons, see the **Black Hole anomaly**.
- We may be able to use the curve matching algorithm to account for these unwanted variations with exposure time.

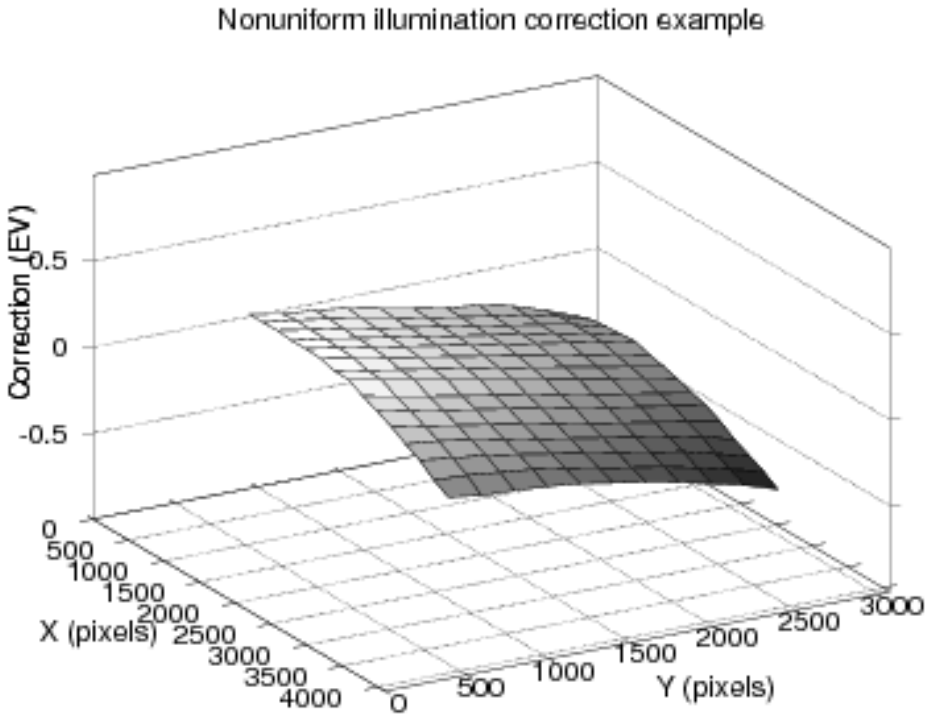
11.5.6.2 Correcting for nonuniform illumination

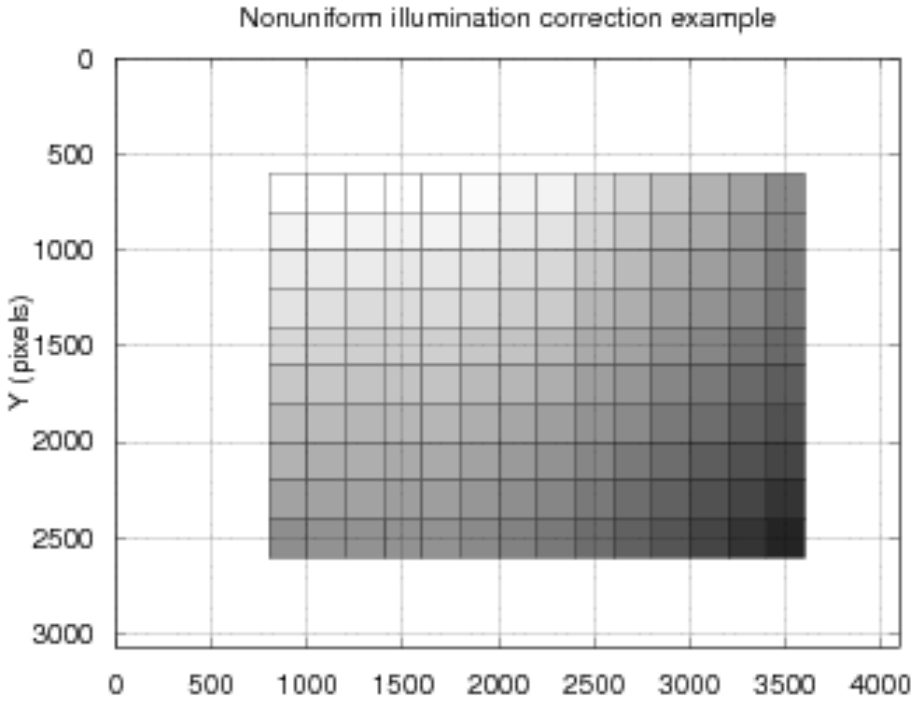
Sampling data from the edge of the chart (left) lets us correct for nonuniform illumination: one can either adjust the chart itself (right), or the reference data (preferred).




1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
A B C D E F G H I J K L
IT8.7/2-1993
2013:10
Charge: R131007

See also:

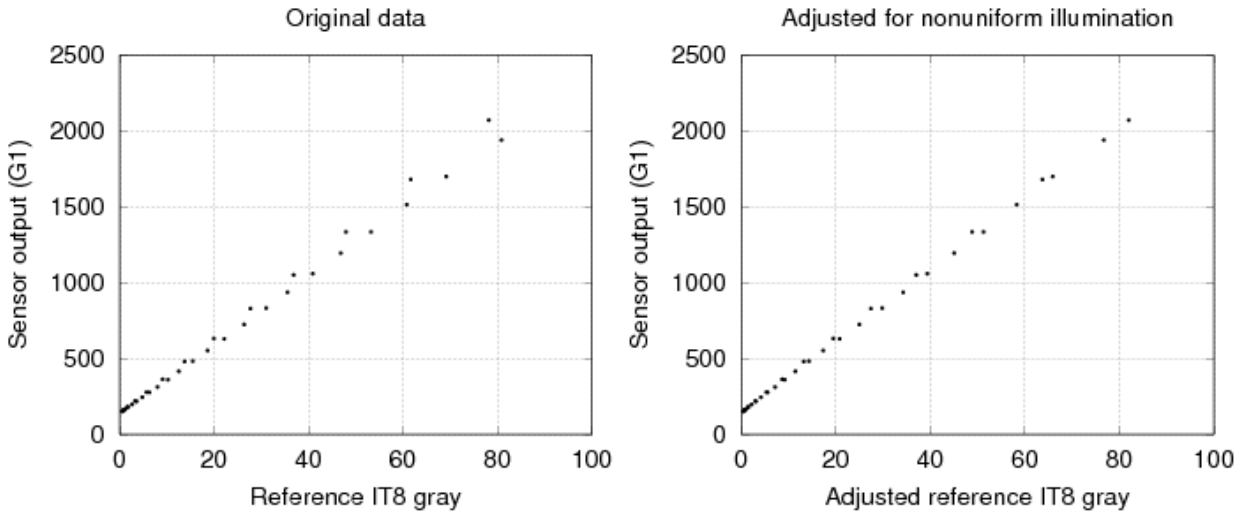

Nonuniform illumination correction example
Correction (EV)
0.5
0
-0.5
0
500
1000
1500
2000
2500
3000
3500
4000
X (pixels)
0
500
1000
1500
2000
2500
3000
Y (pixels)



Why it's better to adjust the reference data instead of chart pixels?

- Black level in our setup is uncertain.
- The chart borders are fairly bright, so the measurements of border brightness are not really influenced by small black level variations.
- Adjusting the image must be done in log space, so we must operate on linearized data.
- If the black level is uncertain, the adjusted dark swatches will have large errors.
- If the response curve is not linear (and not known), adjusting the data under the assumption of linearity will introduce extra errors.
- On the other hand, reference IT8 data is already in linear XYZ space, so adjusting it will not introduce extra errors (other than our less-than-accurate measurements).
- The measurement errors can be hopefully solved with an iterative procedure (anyone able to prove it?).

Plotting raw data vs reference values on IT8 gray swatches gives:



In the left figure, the two grayscale lines from the IT8 chart diverge because of nonuniform illumination. After adjusting the reference data, the match is much better, though it's still not perfect (because we can't really measure the nonuniform illumination in the chart, other than by including it in the model when performing the color profiling step).

11.5.6.3 Iterative procedure for estimating the response curve in the presence of nonuniform illumination

- Initial estimation of response curve, from noisy data (because of nonuniform illumination).
- Adjust the reference data from brightness on chart borders (use griddata to interpolate).
- Estimate the response curve again, this time from much cleaner data.
- Repeat steps 2-3 until convergence.

Unfortunately, this is not going to solve the mismatch between the two scales. Better get some properly illuminated charts :)

12 Associated Use-cases

Overview text required.

12.1 Configuration for Photography

To increase the quality of created images it is highly recommended to do Factory Calibration of your AXIOM Beta first. See 6.3.1.2.

12.1.1 Calibration

Compile raw2dng inside AXIOM Beta

1. Acquire the source files from https://github.com/apertus-open-source-cinema/misc-tools-utilities/tree/master/raw2dng.
2. Copy files to AXIOM Beta.
3. Run `make` inside camera.

Install dcraw inside AXIOM Beta

```
pacman -Sy pacman -S  dcraw
```

Picture Snapping script

Download https://github.com/apertus-open-source-cinema/beta-software/blob/master/beta-scripts/picture_snap.sh

... to camera.

Make it executable:

```
chmod +x picture_snap. sh
```

12.1.2 Operation

Picture snaps are saved in subfolders in the cameras `/opt/picture-snap/` directory.

The script automatically names each subfolder and image by a timestamp: `%Y%m%d_%H%M%S` .

Execute the image snapping by providing the exposure time running eg.:

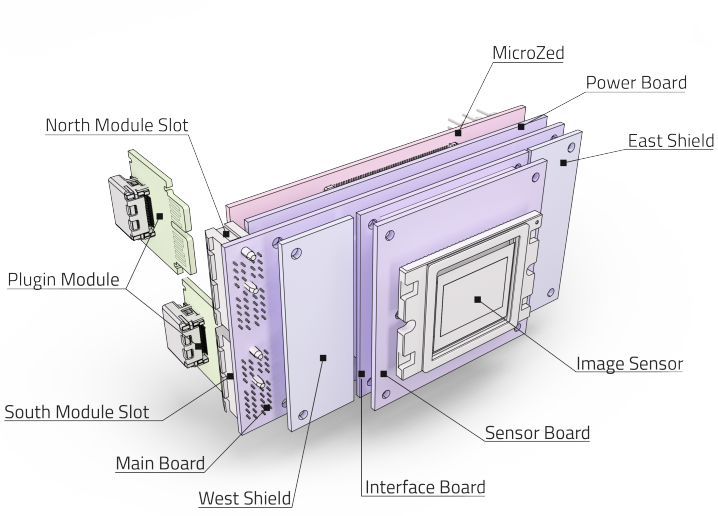
```
./picture_snap.sh 20 ms
```

One complete capture process plus DNG & JPG conversion takes 18 seconds currently.

13 Hardware

Overview text required.

13.1 PCB Stack Layout



Above: Version 5 of the camera's PCB stack layout.

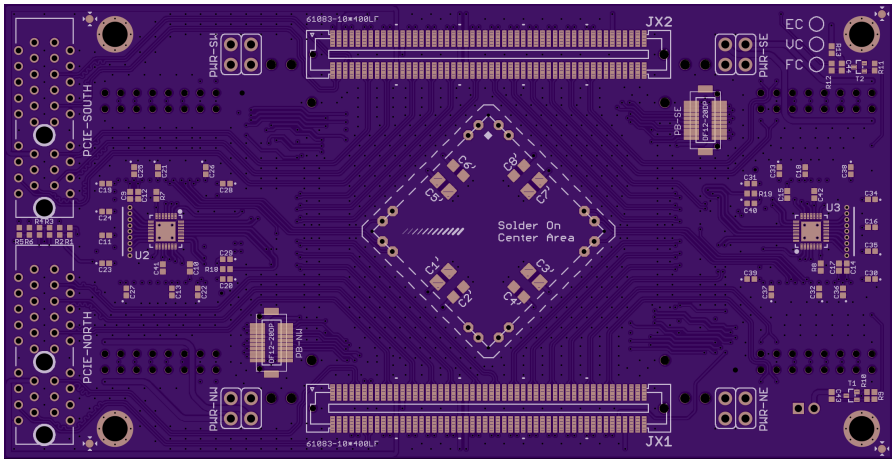
13.1.1 AXIOM Beta CMV12K THT Sensor Board

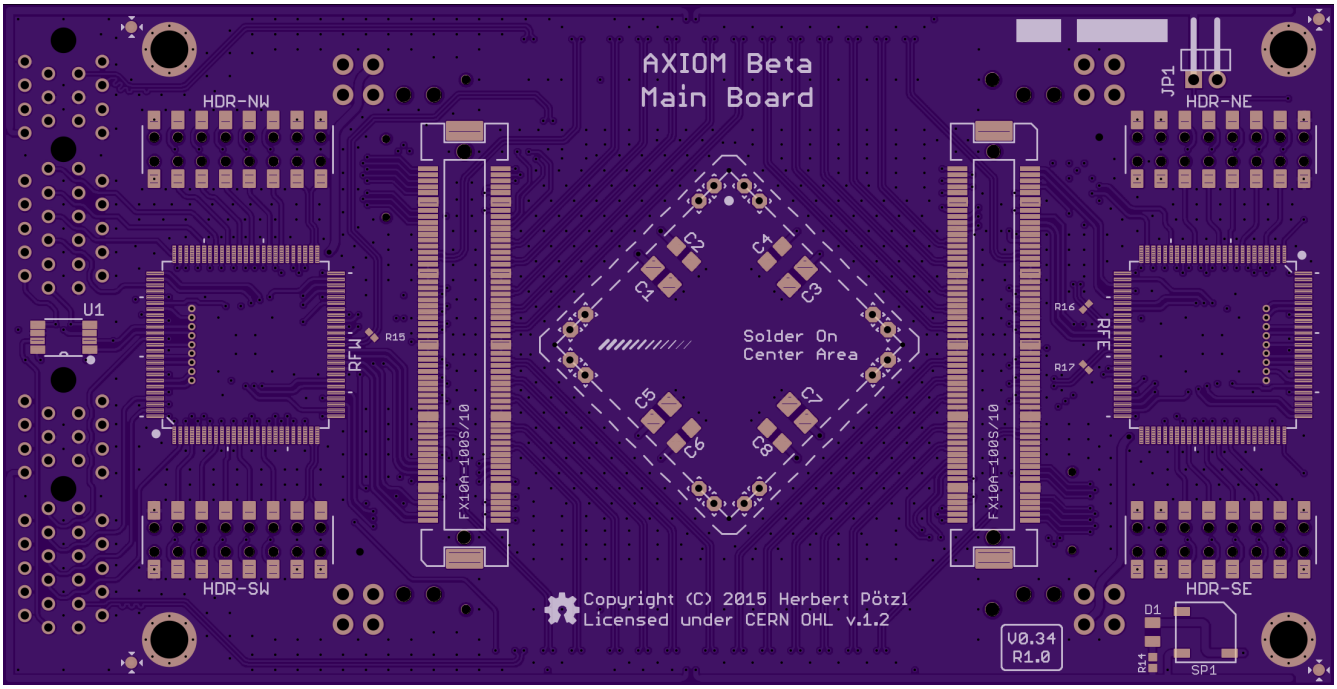
For board revions see - https://wiki.apertus.org/index.php/Beta_CMV12K_THT_Sensor_Board

13.1.2 AXIOM Beta Main Board

The AXIOM Beta Mainboard deals with interfacing the plugin modules and shields and therefore acts as a kind of data crossroad.

The purpose of the two Lattice FPGAs (the so called routing fabrics) is to handle all the low speed GPIO stuff required for plugin modules, shields and CSO without sacrificing valuable Zynq GPIOs.





Beta Mainboard PCB populated with components:

13.1.3 AXIOM Beta Power Board

The AXIOM Beta Power Board PCB is the last board in the stack before the MicroZed. It generates all the different supply voltages for the chips and logic on the other PCBs inside the camera. It also monitors currents so that it can estimate remaining power based on the recorded consumption. In the current revision of the camera a predefined set of supply voltages matching the current application with the rest of the camera have been generated, in the future however, it will be possible for users to dynamically reconfigure voltages according to their needs through the cameras software.
Beta Power Board PCB populated with components:

For board revions see - https://wiki.apertus.org/index.php/Beta_Power_Board.

13.1.3.1 Calibrating Voltages

This describes the process required when factory assembling the power board hardware in the AXIOM Beta stack.

```
./power_init. sh
```

```
./power_on. sh
```

Wait a little...

```
./pac1720_info. sh
```

This will output something like:

```
ZED\_5V          4.8828 V [1f40]         +10.1562 mV [104]   +677.08 mA
BETA\_5V         4.8828 V [1f40]          +1.2891 mV [021]    +85.94 mA
HDN             3.2812 V [1500]         +0.0000 mV [000]     +0.00 mA
PCIE\_N\_V        3.2422 V [14c0]          +0.0000 mV [000]     +0.00 mA
HDS             3.2031 V [1480]         +0.0000 mV [000]     +0.00 mA
PCIE\_S\_V        3.2422 V [14c0]          +0.0000 mV [000]     +0.00 mA
RFW\_V           3.2422 V [14c0]          -0.0391 mV [fff]     -2.60 mA
IOW\_V           3.2617 V [14e0]          +0.0000 mV [000]     +0.00 mA
RFE\_V           3.2422 V [14c0]          +0.0000 mV [000]     +0.00 mA
IOE\_V           3.2812 V [1500]          +0.0000 mV [000]     +0.00 mA
VCCO\_35         2.4219 V [ f80]          -0.0391 mV [fff]     -2.60 mA
VCCO\_13         2.4609 V [ fc0]          +0.0000 mV [000]     +0.00 mA
PCIE\_IO         2.4609 V [ fc0]          -0.0781 mV [ffe]     -5.21 mA
VCCO\_34         2.4609 V [ fc0]          +0.9766 mV [019]    +65.10 mA
W\_VW            2.7734 V [11c0]          +0.0000 mV [000]     +0.00 mA
N\_VW            2.8125 V [1200]          +0.0000 mV [000]     +0.00 mA
N\_VN            2.7734 V [11c0]          -0.0391 mV [fff]     -2.60 mA
N\_VE            2.8516 V [1240]          +0.0000 mV [000]     +0.00 mA
E\_VE            2.6953 V [1140]          -0.0391 mV [fff]     -2.60 mA
S\_VE            2.8516 V [1240]          +0.0000 mV [000]     +0.00 mA
S\_VS            2.7344 V [1180]          -0.0391 mV [fff]     -2.60 mA
S\_VW            2.8516 V [1240]          -0.0781 mV [ffe]     -5.21 mA
```

Now run:

```
watch -n 0.2 ./pac1720_info. sh
```

... which will display the voltages in a clear screen and constantly update the values until you press CTRL+C.

The PCB labels and the labels in the CLI correspond like the following:

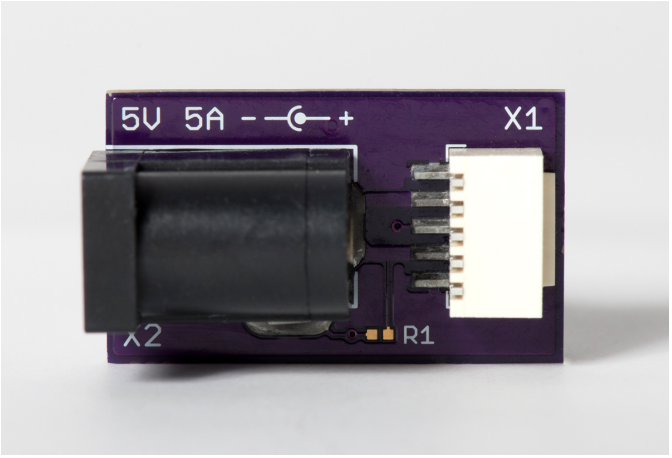
```
WW = W\_VW
NW = N\_VW
NN = N\_VN
NE = N\_VE
SW = S\_VW
SS = S\_VS
SE = S\_VE
EE = E\_VE
```

Use a tiny screwdriver and adjust the trimmers on the PCB until your values look like this:

```
W\_VW            2.4609 V [ fc0]      -0.0391 mV [fff]     -2.60 mA
N\_VW            3.2422 V [14c0]      +0.0000 mV [000]     +0.00 mA
N\_VN            1.8750 V [ c00]      -0.0391 mV [fff]     -2.60 mA
N\_VE            3.2617 V [14e0]      +0.0000 mV [000]     +0.00 mA
E\_VE            3.2812 V [1500]      +0.0391 mV [001]     +2.60 mA
S\_VE            1.9922 V [ cc0]      +0.0000 mV [000]     +0.00 mA
S\_VS            2.9883 V [1320]      -0.0391 mV [fff]     -2.60 mA
S\_VW            1.9531 V [ c80]      -0.1172 mV [ffd]     -7.81 mA
```

13.1.4 Beta Power Adapter Board

Overview text required.

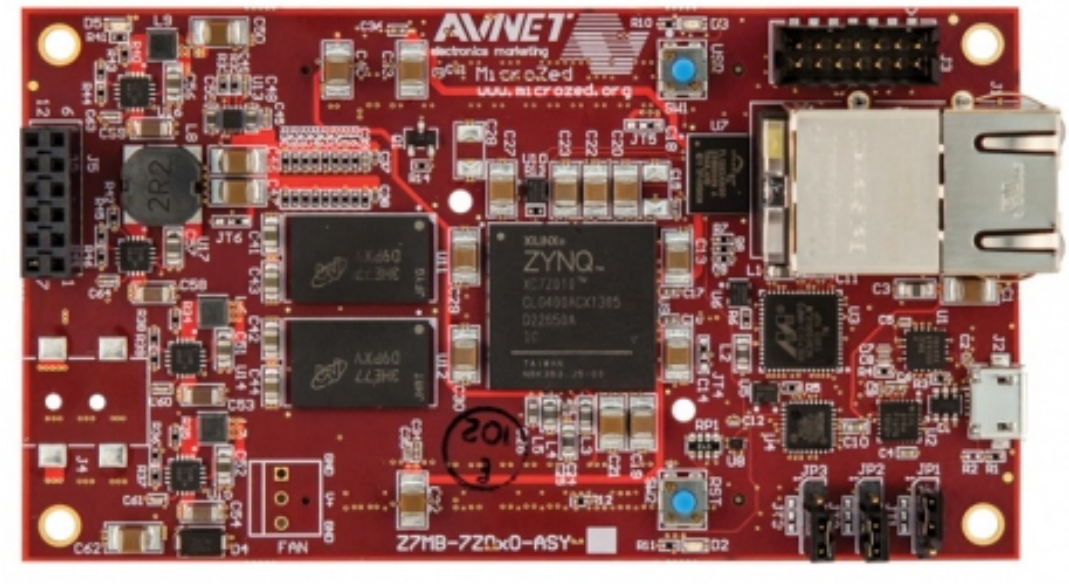


For board revions and concepts see - https://wiki.apertus.org/index.php/Beta_Power_Adapter_Board.

13.1.5 AES-Z7MB-7Z020-SOM-G MicroZed

MicroZed is a low-cost development board based on the Xilinx Zynq-7000 All Programmable SoC. Its unique design allows it to be used as both a stand-alone evaluation board for basic SoC experimentation, or combined with a carrier card as an embeddable system-on-module (SOM). MicroZed contains two I/O headers that provide connection to two I/O banks on the programmable logic (PL) side of the Zynq-7000 All Programmable SoC device. In stand-alone mode, these 100 PL I/O are inactive. When plugged into a carrier card, the I/O are accessible in a manner defined by the carrier card design.

Datashet.



Features:

- SoC XC7Z010-1CLG400C

Memory:

- 1 GB of DDR3 SDRAM
- 128 Mb of QSPI Flash
- Micro SD card interface

Communications:

- 10/100/1000 Ethernet
- USB 2.0
- USB-UART

User I/O (via dual board-to-board connectors):

- 7Z010 Version
- 100 User I/O (50 per connector)
- Configurable as up to 48 LVDS pairs or 100 single-ended I/O

Other:

- 2x6 Digilent Pmod compatible interface providing 8 PS MIO connections for user I/O
- Xilinx PC4 JTAG configuration port
- PS JTAG pins accessible via Pmod
- 33.33 MHz oscillator
- User LED and push switch

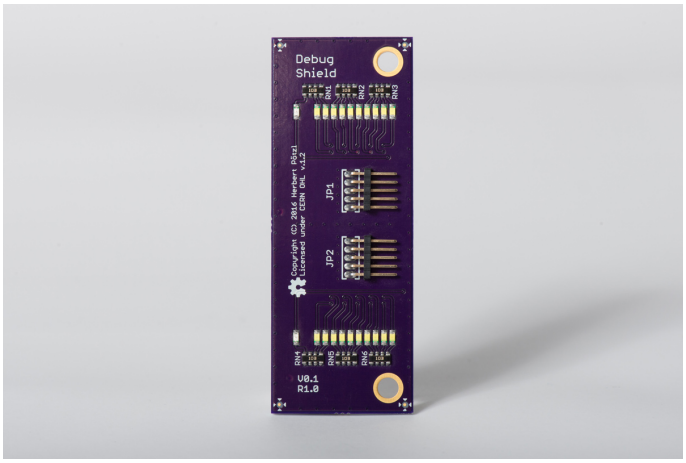
Wiki - https://wiki.apertus.org/index.php/AES-Z7MB-7Z020-SOM-G.

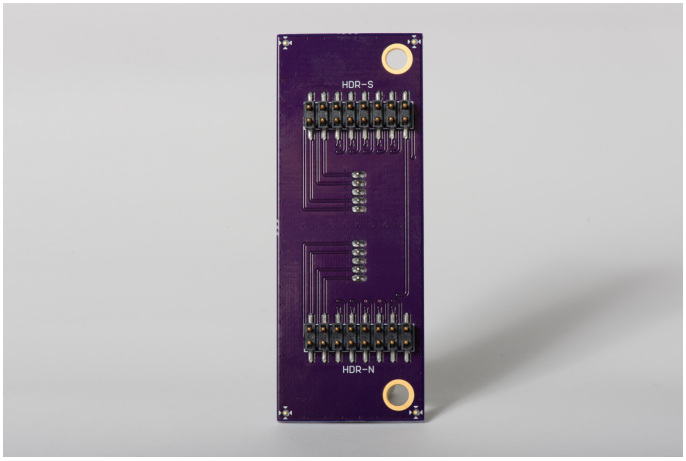
13.1.6 Shields

Overview text required.

13.1.6.1 Beta Debug Shield

2x10 GPIO banks as LED indicators plus two power LEDs. 4 LVDS pairs routed to external connectors JP1/JP2 (2 LVDS plus one GND each).





For board revions see - https://wiki.apertus.org/index.php/Beta_Debug_Shield.

13.1.7 Plugin Modules

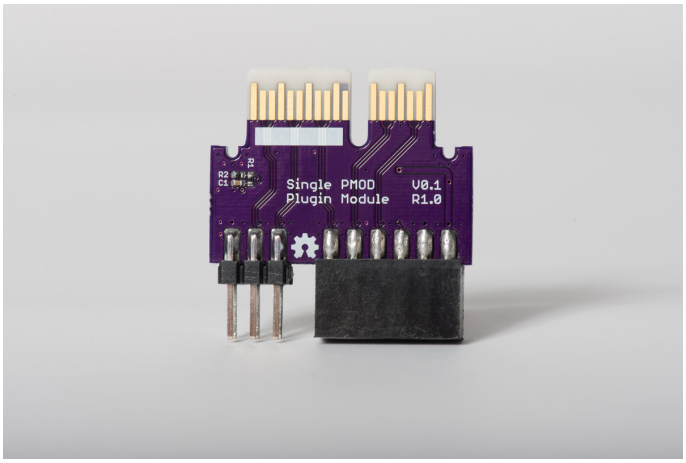
Overview text required.

13.1.7.1 Beta HDMI Plugin Module

This module provides one 1080p60 HDMI output stream via 4 LVDS channels directly from the Zynq on the Microzed
For board revions, dimensions and other files see - https://wiki.apertus.org/index.php/Beta_HDMI_Plugin_Module.

13.1.7.2 Beta 1x PMOD Plugin Module

Overview text required.



For board revions see - https://wiki.apertus.org/index.php/Beta_1x_PMOD_Plugin_Module.

13.1.7.3 4K HDMI Plugin Module

Work in progress.

13.1.8 EEPROM

All PCBS we created for the AXIOM Beta (beside the AXIOM Beta Mainboard) have an onboard EEPROM. These chips provide between 4KB and 16KB of non-volatile memory.

Our primary use for these EEPROMs is to store unique identifiers for each board/type. In a modular stack of PCBs the camera firmware can read the EEPROMs of each PCB and identify which boards, versions and revisions are attached (as different board versions might need different drivers/firmware elements).

13.1.8.1 Proposed Unique ID Structure

2 byte board type identifier (65.536 possible values):

```
00 - AXIOM Beta Powerboard 01 - AXIOM Beta Interface Board 02 - AXIOM Beta Sensor Front
Shield
```

1 byte board output identifier:

```
number of output  interfaces
```

1 byte board active/passive identifier.

4 byte board current draw range identifier

```
2 bytes typical low end range in mA 2 bytes typical high end range in  mA
```

2 byte board length in mm.
1 byte number of used LVDS lanes.
1 byte number of used GPIO channels.
2 byte board identifier:

```
00 - AXIOM Beta Powerboard 01 - AXIOM Beta Interface Dummy 02 - AXIOM Beta Interface Bo
Shield
```

2 byte board version identifier - (ASCII A.BC corresponding to label printed on PCB)

2 byte board revision identifier - (ASCII A.BC corresponding to label printed on PCB)

128 byte UUID

13.2 Power Supply

13.2.1 AC Power Supply

13.2.2 DC Power Supply

13.2.3 Active Battery Mount

13.3 Enclosure

13.3.1 Skeleton

13.3.2 Simple Enclosure

13.3.3 Transparent Acrylic Enclosure

13.4 Optical Information

13.4.1 Lens Mount

13.4.2 Lens Mount Overviews

13.4.3 Infra Red / Ultra Violet Cut-off Filter

13.4.4 Optical Low-pass Filter (OLPF)

14 Support

14.1 Contact Details and Communication Channels

14.2 Regional Communities

14.3 Useful Links

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 00:0a:35:00:01:26 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.9/24 brd 192.168.0.255 scope global dynamic eth0
       valid_lft 172739sec preferred_lft 172739sec
    inet6 fe80::20a:35ff:fe00:126/64 scope link
       valid_lft forever preferred_lft forever
```

```
cd ~/.ssh/
cp authorized_keys authorized_keys.orig
```

ghgffghfg