KERNEL testbench documentation

Frantz

June 22, 2021

Contents

1 introduction

These notes attempt to document the software to configure and drive the hardware used by the kernel test bench. The locations of the different files are only valid for the **kbench** user.

In general, code source files are expected to be placed in sub-directories of the ***home/kbench/Progs*** directory.

2 tmux

Quick notes on `tmux` the terminal multiplexor!

2.1 Create a named `tmux` session

```
tmux new-session -s 'kcam_server'
```

will create a session called "kcam$_{\text{server}}$".

Doing so will result in you **attaching** to this named `tmux` session. You will know you are **attached** to this session because you will see a green bar at the bottom of your current terminal window. If you type `exit`, you will not only leave the `tmux` session, you will also terminate it… which is not what you want to do! What you want to do is **detach** from the session!

2.2 Detach from a `tmux` session

To detach from that session, while leaving it running, you have to use "CTRL-B D": that means keep the "control" and the "b" key together and then type "d" to detach.

2.3 Attach

The following commmand will attach the "kcam$_{\text{server}}$" session that was created earlier.

```
tmux a -t kcam_server
```

3 hardware

3.1 point grey camera

3.1.1 driver set up, API/SDK and configuration tidbits

TBD

3.1.2 control

To run the camera, the following command, lauched from anywhere will start acquisitions with the point grey camera

```
FlyCapSHM
```

3.2 CRED1 camera

3.2.1 setup

The camera is connected to the computer via a VisionLink F4 PCI board by EDT. The driver should be loaded by default when the computer reboots. This can be verified by looking at whether the corresponding kernel module is indeed loaded:

```
lsmod | grep edt
```

After a reboot, the driver must be reminded of the configuration of the camera. The driver installs an initialization program that takes several arguments and a configuration file. A valid configuration file called `cred1_FGSetup_16bit.cfg` for the CRED1 is located in the `~/Progs/camera/kcam` directory hosting the source code controling the camera.

The command to initialize the PCI board is as follows:

```
cd Progs/camera/kcam/ # if you are not already in that directory
/opt/EDTpdv/initcam -u 0 -c 0 -f cred1_FGSetup_16bit.cfg
```

The camera control software expect to be run from inside a tmux session. Tmux is a terminal multiplexer like GNU screen that SCExAO uses and that was therefore adopted here. The benefit of using such a terminal multiplexer is that a program or a job can be launched (such as starting to cool the camera), and then the session detached without interrupting the job.

In the current implementation, two named tmux sessions should be launched:

- `kcam_server` is where the user is going to spend most of its time to interact with the camera and the PCI board
- `kcam_fetcher` is a session where that hosts the continuous acquisition mode of the camera. It is not particularly useful to look at what is going on there most of the time, but it is good to know that it exists if there is a problem.

These named tmux sessions are launched using the following commands:

```
tmux new-session -s 'kcam_server'
tmux new-session -s 'kcam_fetcher'
```

This is a lot to take in, simply to start the camera. This is why in the `~/Progs/camera/kcam/` directory, there is a python script called `kcam_setup` that can be used to start things up with just one command... but it is not well documented YET!

An issue can arise when launching the 'kcam$_{\text{fetcher}}$' tmux session if a shared memory file of the same name already exists. So it is good practise to ensure that if a file exists in "*dev/shm*" called kcam.im.shm, it is deleated. The shm code would previously default to a strange location on a drive if there was a conflict (found deep in the ImageStreamIO library). This was fixed by forcibly updating the variable "\$MILK$_{\text{SHMDIR}}$" whenever a new terminal is launched. A line of code was added to ~/.bashrc to update this variable with the path *dev/shm* .

Currently on the kbench computer this path is updated when any terminal is opened, including when launching the kcam$_{\text{fetcher}}$ tmux session. Thus should work. As it stands, there is no conflict between the kcam$_{\text{fetcher}}$ feed into shared memory, and any fake data stream that can be sent to a shm of the same name. Just be carefull not to have both on at the same time!

To kill a tmux session type:

```
tmux kill-session -t <name>
```

3.2.2 operation

When everything has properly been setup, one can interact with the camera via a command line tool that is launched inside a tmux session named `kcam_server` (cf setup). To connect to that session from your current shell, simply launch:

```
tmux a -t kcam_server
```

which will get you to the camera control shell. The prompt should say `Kcam>`. If it doesn't, that means that either things were not properly setup or that the program was stopped. Neither of these is particularly bad, and the camera shell can be restarted with the following command:

```
./kmcaserver
```

which should bring you the `Kcam>` prompt. This limited camera shell is mostly to control and monitor the behavior of the camera: query for status, control the cooling, set the frame rate, the exposure time... and start the continuous acquisition. The camera offers a lot of options, so refer to the documentation of the camera if you don't know what to do, or use the `help` command inside the shell, that will at least give you an idea of the kind of things that are possible from this command line. The easiest command to start with should be `status`. If everything is OK, the camera responds `ready`.

If the camera has been cooled, and the desired acquisition mode properly setup (refer to the camera documentation), then you can simply hit "start" to trigger the continuous acquisition script. Images should become available on a shared memory data structure and visualized by an external live viewer:

```
shmview /dev/shm/kcam.im.shm &
```

3.3 deformable mirror

3.3.1 driver set up

3.3.2 configuration

After a computer reboot, one needs to remind the driver what kind of DM it is dealing with. A one time init operation is required. After that, the DM electronics should be able to make sense of the commands you are sending its way.

```
cd Progs/DM/bin
./bmc_ltest -I 22
```

3.3.3 control

The control of the DM is an ongoing development. At the moment, the `~/Progs/DM/control/` directory hosts a little utility called `bmc_control`. You have to run it from the command line and give as an argument, the name of a 1024 lines text file that contains (well, the first 507 lines anyways) the commands sent to the 507 actuators.

The current best flat contains the flat provided by BMC (for offset 500 nm) and a small amount of astigmatism correction that was identified after trial and error. To apply this flat, use the following command:

```
cd ~/Progs/DM/control
./bmc_control custom_zernike_flat.txt
```

If you want to do some more sophisticated type of wavefront control, such as introducing controllable amounts of zernikes, you need to look at the `mode_to_command_converter.py` python script that is in the same directory, and look for a way to bend it to your will. Eventually, we'll have the DM run by a server like the SCExAO DM, listening for commands on multiple channels and applying corrections fast.

3.4 Pupil Mask Wheel

The pupil mask wheel sits at the 1st pupil-plane after the DM (Globaly the 3rd pupil-plane). The mask wheel contains a number of various pupil-masks (up to 8 in the existing wheel) which can be placed and aligned to the post-DM pupil, that can also serve as a Lyot-stop if needed. The masks are laser-cut to match the DM geometry 1-to-1 so there is no magnification between the two pupils.

The masks can be aligned remotely using the software commands found in the section below. This is done using two distinct motorized units; a Connex wheel that rotates the masks, and two Zaber linear actuators that move the mount in the Vertical and Horizontal direction (if looking at the pupil, moves the mask up-down and left-right). Note that the focus is not motorised and must be done manualy in the lab, however this should not drift and is very tolerant. Thus by using terminal commands outlined below it is possible to accurately position the pupil mask to the desired rotation and X-Y position.

To monitor the pupil mask position in relation to the DM, the pupil-plane is re-imaged onto the Point-Grey camera (wavelength Vis -> 1 μ m) which shows both planes conjugated, and is this visible in real-time through the pointgrey viewer. [TODO: Should give it a name]

3.4.1 Running the software

The Mask Wheel is entirely controlled from a terminal window, so open one from anywhere. The master command is `maskwheel`, and typing it will bring up a help menu outlining all the possible commands. Below are the commands that can be used with some examples.

```
$  maskwheel 1-9
```

If an integer between 1 and 9 (inclusive) is entered the two zabers and conex wheel will move automatically to the relevant positions which are saved in a calibration file (see JSON Config File). There are 9 different pre-programed positions that the Mask Wheel can be moved to, corresponding to the 8 places in the Mask holder mount, and one more to have a "Blocked" position for easy backgrounds. The exact stepper values for each position (among other things) are held in the calibration file `/home/kbench/.config/kbench_config/kbench_config.json` which is loaded automatically by the program. They can be updated and saved at any time (see `saveaspos` command below), and allows the user to easily switch between different masks without requiring fine re-alignment.

```
$  maskwheel homeall
```

Homes all the stages (Zabers and conex). A "Home" is sending the motors back to 0. This is sometimes required if there is a power outage and the motors lose track of where they are.

```
$ maskwheel parkall
```

This command sends all the stages to their "Park" positions. They are typically the centre of the travel range of the particular stage.

```
$  maskwheel where
```

This command returns the absolute step positions of all the stages of the Mask Wheel and displays them in the terminal.

```
$  maskwheel saveaspos 1-9
```

This function is used to override and save the current placement of the actuators as a new maskwheel "Position" to the config file for future use. For example, lets say you move to position 4 using '`$ maskwheel 4`', but you find that the mask at that position isn't perfectly aligned with the pupil. You can adjust it using the "move" comands until you are happy, then type '`$ maskwheel saveaspos 4`' and save these new positions to the same memory. Thus, if you go back to pos 4 later, it will have the updated position. Note, the software will throw a prompt asking for confirmation before it overrides and will only accept the word "`yes`" to continue (not Y/y). This stops accidental overides to the mask wheel positions.

```
$  maskwheel vrt (or hor or whl)
```

By typing in "`vrt`" or "`hor`" or "`whl`" it will allow the control of individual axes of the Maskwheel. `vrt` controls the vertical stage, `hor` the horizontal, and `whl` the rotation of the Conex wheel. Each of these has the same secondary commands, listed below:

```
$ maskwheel vrt moveto 23600
```

This moves the relevant stage to an absolute step value (in this case step 23,600). Step values can be any integer between 0 and 290,000 for zabers (`vrt/hor`), and 0 and ??? for the Conex (`whl`).

```
$ maskwheel vrt move 2000
```

The `move` command moves the stage in +/- relative step units. So in this example, the stage will move +2000 steps (roughly 6 mm with ~ 3μ m/step. This comand takes both + and - values.

```
$ maskwheel vrt home
```

Homes only this stage axis.

```
$ maskwheel vrt park
```

Parks only this stage axis

3.4.2 Configuration File

The configuration file that maintains a library of pre-set values, including the USB port mapping is held in a JSON located at `/home/kbench/.config/kbench_config/kbench_confi`
If for whatever reason something should happen to the file and a new one needs to be created, a python script (`/home/kbench/Progs/Motors and wheels/create_new_json_database.py`) can be run to re-generate the config file. However, any updated values will be lost unless the python code is itself updated to reflect new values.

[TO-DO] Write instructions on how the config file works. Probably should write another script for easy updating of config…

3.5 Calibration Source

The Calibration Source refers to the box at the edge of the bench that handles the light coming from multiple sources, before it is injected into the bench (using an endlessly single-mode photonic crystal fibre). The box is totaly enclosed to protect the user in the lab from high powered laser light coming from the SuperK (broadband source). Under no circumstances should the box be opened when the SuperK is on without Nick being there!

In addition to the broadband source, the box also contains an alignment laser at $\lambda = 633$ nm. The box can toggle between the two sources either by pressing a little grey button next to the box, or via the software. Further, there is a filter-wheel inside the cal box that is currently filled with Neutral Density filters, which attenuate the light different amounts, alowing you to decrease or increase the flux into cthe bench easily.

The filter Wheel has 6 different positions, number 1 always being empty.

3.5.1 Controlling the CalSource

The master command for the Cal source is `calsrc`. This can be typed in to any terminal. Pressing enter with no further commands will bring up the help menu.

Below are some usefull comands:

```
$  calsrc filt where
```

This ouputs the current position of the filter wheel (1-6).

```
$ calsrc filt 1-6
```

By entering a number after `filt` you can move the filter wheel to the desired position, with the desired filter. What filter is in what slot is written in the help menu which you can also get by typing `$ calsrc filt`.

the command to move the fli command for the thorlabs filt

4 test

```
import numpy as np
print(np.random.randn(10))
```

This is a λ.

$$\Lambda = \frac{\lambda^2}{\Delta\lambda}$$

test 1	test 2	blah
this is the story	10211	12313212131