Table of Contents

TRB Firmware update

1.- Download firmware version from: http://dpnc.unige.ch/~favrey/FASER/TRB/FW/

2.- Follow procedure from: http://dpnc.unige.ch/~favrey/FASER/TRB/FW/FPGA-UpgradeProcedure.txt

Note that the Quartus Prime programmer binary is available in:

```
/root/intelFPGA/19.1/qprogrammer/bin/quartus_pgmw
```

GPIO

File name convention :

- Mx: x=module number
- Tnnn: n= threshold level from 0 to 255
- Axaa: aa= chip address in hexa
- Cz: z = calibration config

Again you have:

- FRB binary data in .daq, and decoded data in CSV associated
- SCT modules binary data from daq file, put in .mod files for led and ledx lines, and with decoded data CSV associated

UniGe GUI

How to install UniGe GUI (Windows)

Preliminary:

You should have the following setup installed :

- 2 downloads : 'GTK# for .NET' (lower button) AND MONODEV 64-bits (upper button) available on http://www.mono-project.com/download/#download-win
- Please also install the latest version of .net ('Download Microsoft .NET Framework' on google)
- USB DRIVER :
    - Download http://dpnc.unige.ch/~favrey/AFE/FW/fx3/UnigeWin7-Driver.zip and unzip to local dir
    - Plug the board into USB3 port and force windows to point to the driver local dir

Installation new application version:

1. Choose zip file for the version you want
2. Unzip it into a local directory

How to get back your old settings (last files/directories):

1. In the New version installed directory: Save 'app-settings-defaults.json' AS 'app-settings.json'
2. Upgrade 'app-settings.json' from the file/directory list found in 'app-settings.json' of the old version

NB :

- `app-settings.json`: file used by the application
- `app-settings-defaults.json`: best template settings file compatible with the current application (e.g. with new parameters)

How to install UniGe GUI (LINUX)

PRELIMINARIES

First be sure to update your Linux distrib! I'm using Ubuntu 16.04 LTS.

INSTALL MONODEVELOP (only once on a dedicated machine)

There is now a package that install mono develop and all its dependencies, including mono and gtk-sharp. 'mono' and 'gtk-sharp' (on Ubuntu, the 'monodevelop' package depends on both of them, so you just need to install that one; on other distros, your mileage may vary) It's installing a lot more stuff than what we need, but it makes it a lot simpler too:

- `sudo apt-get update`
- `sudo apt-get install monodevelop`

INSTALL USB (only once on a dedicated machine)

You'll find the file in: http://dpnc.unige.ch/~favrey/AFE/SW/Linux/InstallUSB/ (or lib/UnigeFrontEnd/thirdparty/InstallUSB for SVN developper)

Install libUSB: `sudo apt-get install libusb-1.0-0-dev`

NB: with centOS, the distro provides both libusb (version 0.1.5, which is actually the new name, but it's stuck at an older version) and libusbx (version 1.0, which is the older name, but newer version), a different name than expected. I think libusbdotnet looks for the 1.0 version, so you just have to install the development package for that: `# yum install libusbx libusbx-devel`

As root, create a file called 89-bmfeb.rules in /etc/udev/rules.d/ containing this one line to make sure that the USB device is accessible by a non-privileged user: SUBSYSTEMS=="usb", ATTRS{idVendor}=="206b", GROUP="wheel", MODE="0660"

(this will work if the user is part of the "wheel" group, which is the default group for administrators in most linux distros. You can check by running 'groups' from a terminal. You can modify it with a different group name if needed)

Reboot the machine or run as root 'udevadm control --reload-rules && udevadm trigger' to force the computer to reload the device rules.

To check that the board was connected to a USB3 host, you can run "=dmesg | grep xhci=" from a terminal. You should see a line in the output similar to:

[78125.235770] usb 2-2: new SuperSpeed USB device number 12 using xhci_hcd

this means the new USB device is recognized as a SuperSpeed device using the right kernel module.

APPLICATION

1. Choose zip file for the version you want
2. Unzip it into a local directory
3. Launch the application with: `sudo mono UnigeFrontEnd.exe`

For example:

`/home/faser/Software/UnigeGUI-v841`

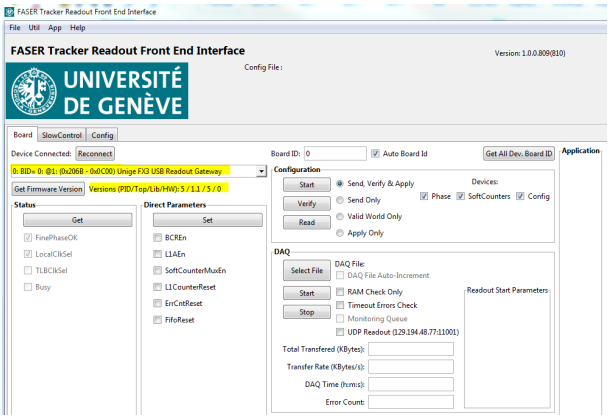
How to get back your old settings (last files/directories):

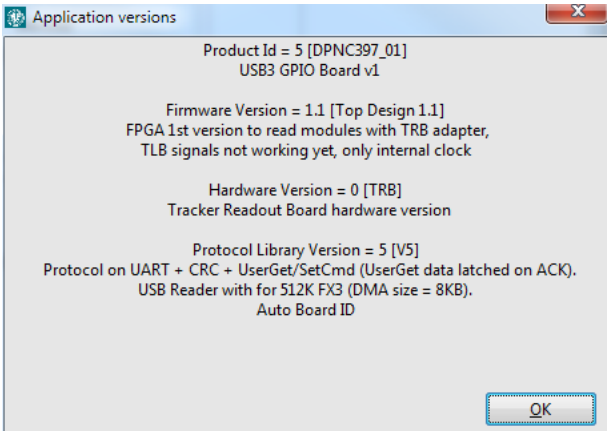
- In the New version installed directory: Save 'app-settings-defaults.json' AS 'app-settings.json'
- Upgrade 'app-settings.json' from the file/directory list found in 'app-settings.json' of the old version

NB : * `app-settings.json`: file used by the application * `app-settings-defaults.json`: best template settings file compatible with the current application (e.g. with new parameters)

GUI test

- Launch the GUI
- Below Device Connected , you should see Unige FX3 USB Readout Gateway
- Then Push button Get Firmware Version , you should have 5/1.1/5/0





How do set L1 delay in scan script

In the TRB spec., the way it has been chosen to do the calibration command pulse and the L1A trigger after a certain number of clock ticks is the following :

- You have to send the command which is Field3 = 0x0C and Field5 = 0x30 (constant _0C_F5_CalPulse) and Field6_0 _9
- When the FPGA receive this specific command, it will read the Field6 sub-fields which can be up to 10x16-bits (Field6_0 up to Field6_9) which should contain series of b 0 and b 110 at the end. One 0 = 1 clock tick delay, and the b 110 at the end stops the FPGA sending s train of pulse since this corresponds to a L1A trigger.

Field6_0 is used by many commands (see table attached) Field6_0 to _7 is used by Mask Register Field6_0 to _9 is used by Issue Cal Pulse and L1A :

If you want 113 clock ticks, then you need (7x16 + 1) x b 0 sent : => Field6_0 to Field6_6 = 0x0000 => Field 6_7 = b 0110 0000 0000 0000 = 0x6000 Others Field6_8 to _9 will not be read but better to put them at 0x0000

If you want 100 clock ticks for instance, then you need (6x16 + 4x 0) sent : => Field6_0 to Field6_5 = 0x0000 => Field 6_6 = b 0000 1100 0000 0000 = 0x0C00

All possible values for Field6_x are:

- Delay : 0x0000
- L1A trigger : 0x0003, 0x0006, 0x000C, 0x0018, 0x0030, 0x0060, 0x00C0, 0x0180, 0x0300, 0x0600, 0x0C00, 0x1800, 0x3000, 0x6000, 0xC000
- But also Field6_x=0x0001 and Field6_x+1=0x8000 (L1A shared between two 16-bits Field6_x)

tcalib

`tcalib` is the main application to run the calibration of the Faser tracker planes. It is a command-line executable within the `apps` directory of the TrackerCalibration gitlab repository (https://gitlab.cern.ch/faser/trackercalibration). The calibration relies in a series of so-called scans, where typically a fixed calibration charge is injected into the readout circuitry and then one parameter of the FE electronics is varied / scanned. Some examples are the so-called ThresholdScan where the discriminator is varied or the StrobeDelay scan, where the strobe delay parameter is varied.

Usage and command-line arguments

The usage is very simple:

```
./tcalib -i <JSON_FILE> -t <TEST_IDENTIFIER> [OPTION(s)]
```

where:

- `JSON_FILE` is an input configuration file (in JSON format), corresponding for the moment to a single SCT module or to a full-plane. Some examples are given below.
- `TEST_IDENTIFIER` is a number (or a concatenation of numbers) used to identify one or more of the tests listed below:

Code	Test	Comment
1	Mask scan	
2	L1 delay scan	
3	Threshold scan	
4	Strobe delay scan	
5	3-point gain	
6	Response curve	
7	Trim scan	
8	Noise-occupancy scan	not finalized

Note that both the input configuration file and the test identifier(s) are required arguments.

Some options to the `tcalib` binary are:

Option	Alternative	Parameter	Description	Default
`-o`	`--outBaseDir`	`OUTDIR`	Base output results directory	`/home/shifter/cernbox/b161`
`-v`	`--verbose`	`VERBOSE_LEVEL`	Sets the verbosity level (from 0: minimal, to 3: maximal)	0
`-d`	`--l1delay`	`L1DELAY`	delay between calibration-pulse and L1A	130
`-e`	`--emulate`	-	Emulate TRB interface	false
`-n`	`--noTrim`	-	Do not load in chips the single-channel trimDac values	false
`-h`	`--help`	-	Displays help information and exit.	

Since the loading the trimDac values is slow (for every channel one needs to send a command with the corresponding TrimDac value), the `-noTrim` option is useful to save time in the chip configuration in situations where channel trimming is not required (e.g. code debugging, checking electrical functionality of a module, testing various configurations, etc). In particular, when testing a new module or plane, no trim is required at the beginning.

Note the tests can be concatenated to the `-t` argument. This is particularly useful to launch a full automatic plane calibration with a test-sequence consisting in a series of scans run in the right order. For example, the command

```
./tcalib -i config/Module1.json -t 345
```

will run on a module described by the configuration file `config/Module1.json` (relative to the current path), first a Threshold scan (test-code 3), then a StrobeDelay scan (test-code 4) and finally a Three-point gain test (test-code 4).

Configuration files

Single-module configuration files

A minimal example of single-module json configuration file is shown below.


```
{
    "Chips": [
        {"Address": 32, "BiasDAC": 6169, "ConfigRegister": 8192, "StrobeDelay": 0, "Threshold": 0
        {"Address": 33, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 34, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 35, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 36, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 37, "BiasDAC": 6169, "ConfigRegister": 4096, "StrobeDelay": 0, "Threshold": 0
        {"Address": 40, "BiasDAC": 6169, "ConfigRegister": 8192, "StrobeDelay": 0, "Threshold": 0
        {"Address": 41, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 42, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 43, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 44, "BiasDAC": 6169, "ConfigRegister": 2048, "StrobeDelay": 0, "Threshold": 0
        {"Address": 45, "BiasDAC": 6169, "ConfigRegister": 4096, "StrobeDelay": 0, "Threshold": 0
    ],
    "PlaneID": 1,
    "ID": 20220380200206,
    "TRBChannel": 1
}
```


The file basically contains an array of chip objects, with:

- `Address`: chip address (decimal value)
- `BiasDAC`: contents of bias register (decimal)
- `ConfigRegister`: contents of configuration register (decimal)
- `StrobeDelay`: value of strobe-delay (decimal, DAC range [0; 63])
- `Threshold`: threshold value (decimal, DAC range [0; 255])

Note that:

- the field "TRBChannel" is the channel in the TRB to where the module is connected.
- the fields `"PlaneID"` and `"ID"` are somewhat undefined (and finally not really necessary to run the calibration since for both histogramming and command sending `TRBChannel` is used). However, we think it is useful to use them to include useful information, such as the plane identifier the module belongs to (`"PlandID"`) or the module serial-number (`"ID"`).

Tracker plane configuration file

An example of json configuration file for a plane with 8 SCT modules is shown below. It is a simple file just containing the paths to the single-module config files, either as relative paths to the plane cfg file or as absolute paths.

```
{
    "Modules" : [
        {"cfg" : "Module0.json"},
        {"cfg" : "Module1.json"},
        {"cfg" : "Module2.json"},
        {"cfg" : "Module3.json"},
        {"cfg" : "Module4.json"},
        {"cfg" : "Module5.json"},
        {"cfg" : "Module6.json"},
        {"cfg" : "Module7.json"}
    ]
}
```

channel masking

By default, if nothing is specified in the module configuration file, no channel is masked. We can specify which channels are masked in each chip with the inclusion of an additional field `"StripMask"` :

```
{
    "Chips": [
        {"Address": 32, "BiasDAC": 6169, "ConfigRegister": 8192, "StrobeDelay": 0, "Threshold": 0
...
}
```

where each value of the `StripMask` array is a 16-bit word representing the status of 16 channels. Note the ordering is important:

- the words in the array are ordered in a way such as the first word corresponds to channels [0; 15], the second word to channels [16; 31] and so on.
- in each word, the LSB is right-most

For example, if a chip has channels 5, 10, 28, 57, 100 and 115 dead / masked, the corresponding mask array in the json configuration file would be:

```
"StripMask" : [1056,  4096, 0, 8192, 0, 0, 16, 8]
```

since effectively:

decimal	hex	binary	meaning
1056	0x0420	0b0000 0100 0010 0000	channels 5, 10
4096	0x1000	0b0001 0000 0000 0000	since channel 28 % 16 = 12
0	-	-	-
8192	0x2000	0b0000 0010 0000 0000	since channel 57 % 16 = 9
0	-	-	-
0	-	-	-
16	0x0010	0000 0000 0001 0000	since channel 100%16 = 4
8	0x0008	0b0000 0000 0000 1000	since channel 115%16=3

`decodeCfg` utilitty

The `decodeCfg` command-line utility takes as input a json configuration file and prints in screen some decoded information:

- all registers are shown in hex
- bitstream of the configuration register, from which it is also shown the state of the edge bit and mask bit

This mostly useful to check single chip configuration, for which verbosity option has to be set. For example:

```
% ./decodeCfg -i Module1.json -v
MODULES : 1

   * mod [0] : id=20220380200206  planeID=1  trbChannel=1  moduleMask=0x02 (0000 0010)  Nchips=12
     - chip [00]  add=32 (0x20)  cfg=0x2000 (0010 0000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [01]  add=33 (0x21)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [02]  add=34 (0x22)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [03]  add=35 (0x23)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [04]  add=36 (0x24)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [05]  add=37 (0x25)  cfg=0x1000 (0001 0000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [06]  add=40 (0x28)  cfg=0x2000 (0010 0000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [07]  add=41 (0x29)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [08]  add=42 (0x2a)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [09]  add=43 (0x2b)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [10]  add=44 (0x2c)  cfg=0x0800 (0000 1000 0000 0000)  biasReg=0x1819  threshcalReg=0
     - chip [11]  add=45 (0x2d)  cfg=0x1000 (0001 0000 0000 0000)  biasReg=0x1819  threshcalReg=0
```

Note the above information might be changed / expanded in the future. For example, it the verbosity level is further increases, trim values per channel will be shown.

Calibration sequence

In order to run the full calibration of a full plane:

```
tcalib -i config/Plane1/Plane1.json -t 14567 -v -n -o /home/shifter/cernbox/b161/Plane1
```

which will run in sequence:

1. Mask scan
2. StrobeDelay scan
3. ThreePointGain test
4. ResponseCurve
5. Trim scan

for the plane described by the json configuration file `config/Plane1/Plane1.json`. The results of all tests will be stored in the base output directory, `/home/shifter/cernbox/b161/Plane1` where subdirectories for each test in the sequence will be created. Finally, configuration files with updated values after the tests (particularly the optimum strobe-delay values per chip and the single-channel trim corrections) will be also created.

-- YosukeTakubo - 2019-10-02

This topic: FASER > TrackerReadout
Topic revision: r10 - 2020-02-27 - SergioGonzalez