JRA1 Telescope: NI Flex RIO DAQ

DAQ emulator software overview

Gilles Claus¹, Mathieu Goffe¹, Kimmo Jaaskelainen¹, Cayetano Santos¹, Matthieu Specht¹

January 17, 2011

Abstract

The EUDET JRA1 Pixel Telescope is using a custom-made data acquisition system since a couple of years. In preparation for AIDA, the group decided to investigate different off the shelf I/O systems. The advantage of such a system is the easier support and the availability over the next years. The IPHC group selected the NI Flex Rio system and prepared LabView sources, which can rather easy be connected to the existing DAQ. In this memo describes the DAQ emulator software which can be used for DAQ development without the hardware.

¹IPHC, Strasbourg, France
# Table of Contents

NI Flex RIO DAQ 1

1 Introduction 4

2 How to compile the software 6

3 How to start the software 13

4 Software GUI overview 16

4.1 Window to log errors and general messages 17

4.2 Application initialization sub panel 19

4.3 Run control sub panel 20

4.4 Load Run sub panel 21

4.5 Run RIO board Emulation sub panel 22

4.5.1 Check frames content subpanel 26

5 Procedure to start emulation 28

6 Playing with the DAQ emulator 30

6.1 Mode EUDET 1 30

6.1.1 Fixed frame size 30

6.1.2 Maximum frame size 35

6.1.3 Random frame size 36

6.2 Mode EUDET 2 37

6.2.1 Default frame size & no trigger 37

6.2.2 Default frame size & one trigger 38

6.2.3 Default frame size & five triggers 39

6.2.4 Default frame size & ten triggers 40

6.2.5 Print result. 41

6.3 Mode EUDET 3 42

6.3.1 Default frame size & no trigger 42

6.3.2 Default frame size & one trigger 43

6.3.3 Maximum frame size & five triggers 44

6.3.4 Maximum frame size & ten triggers 45

6.3.5 Print result. 46

6.3.6 Maximum frame size & 1 trigger / 100 frames 47

6.3.7 Print result. 48

6.3.8 Maximum frame size & 3 consecutive triggers / 100 frames 49

6.3.9 Print result. 50

6.3.10 Random frame size & 3 consecutive triggers / 100 frames & save to disk 51

6.3.11 Load a run from disk 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>How to interface emulator to EUDET DAQ?</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>Introduction</td>
<td>55</td>
</tr>
<tr>
<td>7.2</td>
<td>The input side Run control &amp; Start emulation</td>
<td>56</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Run control context record and configuration function</td>
<td>56</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Emulation context record and configuration function</td>
<td>60</td>
</tr>
<tr>
<td>7.2.3</td>
<td>How to access to context records which variables?</td>
<td>64</td>
</tr>
<tr>
<td>7.3</td>
<td>The output side Telescope data stream</td>
<td>65</td>
</tr>
<tr>
<td>7.3.1</td>
<td>How the Flex RIO board is read?</td>
<td>65</td>
</tr>
<tr>
<td>7.3.2</td>
<td>How the data stream is organized?</td>
<td>69</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Organization of one frame EFRIO_TFrame</td>
<td>70</td>
</tr>
<tr>
<td>7.3.4</td>
<td>The frame header record EFRIO_TFrameHeader</td>
<td>72</td>
</tr>
<tr>
<td>7.3.5</td>
<td>The data part EFRIO_TFrameData</td>
<td>73</td>
</tr>
<tr>
<td>7.3.6</td>
<td>The trigger record EFRIO_TFrameData</td>
<td>77</td>
</tr>
<tr>
<td>7.3.7</td>
<td>The trigger record items</td>
<td>78</td>
</tr>
<tr>
<td>7.3.8</td>
<td>How to access to frames data which variables?</td>
<td>80</td>
</tr>
<tr>
<td>7.4</td>
<td>How / where to write the code?</td>
<td>81</td>
</tr>
<tr>
<td>7.4.1</td>
<td>The eudet_frio library and DLL</td>
<td>81</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Run control context record and configuration function</td>
<td>81</td>
</tr>
<tr>
<td>7.5</td>
<td>Warning about files library</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>Acknowledgement</td>
<td>82</td>
</tr>
</tbody>
</table>
1 Introduction
The DAQ emulator software is a windows application which emulates the data stream produced by a telescope equipped with six Mimosa 26, read by the Flex RIO board. It has been developed with Borland C++ Builder 6 IDE, therefore you need this software suite to compile the project.

Its main goal is to have a tool to test the Flex RIO lib (eudet_frio) without the need to run all the hardware: Mimosa 26, JTAG control, Flex RIO board and DAQ. Because; otherwise debugging will quickly become painful, due to the number of things we need to handle.

Moreover, our EUDET collaborators will have to implement interface from EUDET DAQ to Flex RIO DAQ via Ethernet in this library. It will be easier for them to work with an emulator rather than with the whole DAQ chain (HW & SW).

We should also keep in mind that the DAQ application is Labview and it uses the eudet_frio lib embedded in a DLL, therefore it will complicate debugging. Especially if the software crashed ... Because we don't have an integrated debugger for Labview and the DLL (one day I must take the time to have a close look to LabWindows CVI ;-).

The DAQ emulator also helps in this case, because it doesn't need to see eudet_frio library as a DLL. Library source files can be included in application and therefore we can use the debugger: inspect variable, set breakpoints and so on.

But please don't ask me how to use Borland debugger, I never use it ... I have my own set of macros to log what happens in source file and it's enough in case of problem.
What this DAQ emulator can do?

- It can emulate one or six Mimosa 26
- It has four modes of board “readout” emulation
  - IPHC system → Data format for compatibility with our previous DAQ
  - EUDET 1 → Acquire all frames & but doesn’t store TLU triggers
    It stores only first three triggers → like in the IPHC mode
  - EUDET 2 → Acquire all frames & store TLU triggers (up to 288 /
    frame )
  - EUDET 3 → Acquire only frames with trigger & store TLU triggers

There is only one mode useful for EUDET collaboration → mode EUDET 3. The others modes had been developed to test the software step by step and to compare execution times. Because it was important to evaluate the execution time of frames with trigger extraction by software, and to do this we need to know execution time without this processing.

- The values of Mimosa 26 frame “relevant fields” are configurable from GUI
  - Header
  - Data length
  - Trailer
- The frame counter is incremented automatically, two modes are available
  - By default, incrementation starts at acquisition beginning → 0 .. 1799
  - It can be modified (source code) to increment from beginning of run
- The data size can be configured as
  - Fixed value hard coded in emulation function
  - Random value
  - Maximum value on first Mimosa 26, others hard coded in emulation function
- The data part of the frame contains 0, but user can modify the emulation function code to set any other value.
- Triggers (TLU & Flex RIO) are also configurable
  - The number of trigger to emulate per frame or each N frames
  - The first three triggers + the last one are configurable from GUI
  - The others triggers are hard coded to 0 in emulation function
- Save run to disk, load run from disk, scan run & display frames “relevant fields”
2 How to compile the software

The application source code is in directory x:\prj\win\eudet\emul_flex_rio_daq.

Launch C++ Builder
Open project

by selecting emul_flex_rio_daq.bpr file.
You should get the following window.
Lets have a look on the application source files list:

- **WinMain.cpp** → Main window source code
- **app.inc** → Includes all source files ("copy / paste" of source)
- **app.int** → Includes all interface files (cst, types, etc definitions)
- **app.typ** → Types definitions
- **app.var** → Global variables definition
- **app.h** → Functions prototypes
- **app.c** → C source code

This organisation of files *.def, typ, var, h, c is not proposed or defined by Borland, it’s my way of programming, which in fact comes from Borland Pascal language.
Disable warnings for compilation ...

Compilator panel, subpanel “warnings”
Compile the project

You should get "0 errors"
You can run it from the IDE, and of course can use debugger if needed to set breakpoints, inspect variables and so on. But first, you need to check that parallel port is disabled, otherwise you can’t run the software from IDE.

The compilation directive “EFRIO_INCLUDE_PARA_PORT” must be disabled.

```c
// Disable // port handling & JTAG COM interface
#define EFRIO_INCLUDE_PARA_PORT 0
#define EFRIO_INCLUDE_JTAG 0
```

Now you can run it by a click on the green arrow.
3 How to start the software

You can run it from IDE (see § 2) if you need the debugger, or as a standalone application as explained in this chapter.

You can launch the executable file emul_flex_rio_daq.exe from x:\bin directory.

The following window should appear.
The emulator can also allow you to measure functions execution time by generating a pulse on PC parallel port during their execution. If you want to run the software in this mode, you need to compile it with parallel port enabled (the conditional compilation directive EFRIO__INCLUDE_PARA_PORT must be enabled in file app.def) and to run it via the batch file names: run emul_flex_rio_daq.bat.
If, by mistake, you run the version compiled with PC parallel port handling directly via a call to executable file (not via the batch) you will get exception errors. Because access to parallel port is not allowed, please kill the program and start it via batch file.
4 Software GUI overview

The software has two panels: Main and a Debug panel on which user can add GUI controls and indicators to test code. There is also a Window to log messages.

<table>
<thead>
<tr>
<th>Main panel sub panels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application initialization</td>
</tr>
<tr>
<td>Run control</td>
</tr>
<tr>
<td>Load Run</td>
</tr>
<tr>
<td>Run RIO board Emulation</td>
</tr>
<tr>
<td>Check frames content</td>
</tr>
</tbody>
</table>

The main panel has five sub panels:

- Application initialization
- Run control
- Load Run
- Run RIO board Emulation
- Check frames content
4.1 Window to log errors and general messages

Open this window via the menu “Windows”, submenu “Show Log Window”.

The log level specified is “All”, it means that all kind of messages will be logged.

The following window will appear

Left part displays errors message and right part general messages. The messages are also written in log file specified in the “Application initialization” subpanel.
A closer look to error messages list shows three messages printed as demonstration of messages logging macros.

The letter indicates the level of error and the macro used to print it:

- \#T → Code tracing message → Macro err_trace (…)
- \#W → Warning messages → Macro err_warning (…)
- \#E → Error message → Macro err_error (…)

This is the source code which calls these macros:

```c
// Error messages demo

err_trace ( I ERR_OUT, "This is a trace message from FrmMain - VMyVar=10", VMyVar );
err_warning ( I ERR_OUT, "This is a warning message from FrmMain - VMyVar=10", VMyVar );
err_error ( I ERR_OUT, "This is an error message from FrmMain - VMyVar=10", VMyVar );
```

These macros work like the "old" printf (…), accept the same syntax. In log files they print more information than in GUI source file, function, line number

There is also macros to log general message (right panel):

```c
// General messages demo

msg ( I MSG_OUT, "This is a general message from FrmMain - with default LogLv = 1 - VMyVar=10", VMyVar );
msg1 ( I MSG_OUT, "This is a general message from FrmMain - with LogLv = 2 - VMyVar=10", VMyVar );
```
4.2 Application initialization sub panel

Via this panel you can define

- The error logging level which can be
  - No
  - All
  - Warnings & Errors
  - Errors

- The general message logging level
  - 127 to get all messages
  - Other value depend on the level convention used in macro call

- The log files used to store messages
  - Errors log file ➔ for errors
  - Messages log file ➔ for general messages

Errors and general message are displayed in “Errors & Messages” window (see § 4.1) and printed in log files.

Once you have defined errors and messages log levels, you can click on “Init” button to initialize library and have a look to messages in the log window.
4.3 Run control sub panel

Via this panel you configure the run control:

- **Number of Mimosa 26** → Only two options: 1 or 6
- A number to identify the run
- The total **event number** to store in the run
- The **events number** stored per run file (a run is split in many files)
- The frames number per acquisition
- The **data transfer mode** (IPHC, EUDET 1, 2, 3 → See Introduction)
- Destination directory for run file
- **Run** file name prefix (RUN_666 → RUN_ is the prefix)
- Selection to save or not data to disk
- Selection to send data or not on Ethernet + % of triggers / events sent

Some of these parameters are not handled now, but they will be useful later.

Perform run configuration by clicking on button “Conf RUN”. You can also print run configuration parameters record in log file via button “Print run parameters”.

- 20 -
4.4 Load Run sub panel

Via this panel you specify the run file you want to load:

- The number which identifies the run
- The run directory
- The run file prefix

Once parameters are set, click on “Load” to load a run file, the status field will indicate the result of operation.

Before loading another run or before closing the application, please click on “Close” button.
4.5 Run RIO board Emulation sub panel

Via this panel you configure DAQ emulation parameters

- **Acq cycle [ms]** → Period between two acquisitions
- **Emule DRAM** → A delay to emulate Flex RIO DRAM access (not very useful)
- **Emulation function No** → Select which emulation function to use it is not implemented now, there is only one emulation function.
- **Header** → Header of each Mimosa 26
- **Trailer** → Trailer of each Mimosa 26
- **Trig nb** → Number of triggers / frame
  - In mode IPHC, EUDET 1 → field ignored → Always 3 triggers
  - In mode IPHC, EUDET 1-2 “Trig nb” are emulated on each frame.
  - In mode IPHC, EUDET 1-2 “Trig nb” are emulated on each frame. In mode EUDET 3 it is possible to emulate “Trig nb” each N
frames on $F$ consecutive frames via the fields "Trig on 1 frame/n",
"Trig on $F$ frames".
• Trigger [0], [1], [2], [last] = TLU triggers (up to 288 / frame)
  - In mode IPHC, EUDET 1: They are ignored, three triggers are generated with values 16, 32, 64 for IPHC mode and 1, 2, 4 for EUDET 1 mode.
  - In modes EUDET2, EUDET 3 they allow to specify the first three triggers values [0], [1], [2] and the value of the last trigger [Trig nb – 1]. The triggers between [2] and [Trig nb – 1] have their value set to 0.

• Time stamp [0], [1], [2], [last] = Flex RIO triggers (up to 288 / frame)
  - In mode IPHC, EUDET 1: They are ignored → no time stamp
  - In modes EUDET2, EUDET 3 they allow to specify the first three time stamps values [0], [1], [2] and the value of the last time stamp [Trig nb – 1]. The time stamp between [2] and [Trig nb – 1] have their value set to 0.

• “Trig on 1 frame / N” & “Trig on F frames” are only enabled in mode EUDET 3 and allow to generated “Trig Nb” triggers on F consecutive frames each N frames.
- Random data size → Allows to generate random data size on each Mimosa 26, by default the data size is fixed and hard coded in the emulation function.

- Max data size → Set maximum data size on first Mimosa 26

This panel has also displays

- **Acquisition counter** → Counter of acquisitions

- **Events counter** → Acquisition counter X Frame nb per

- **Acq funct return size** → Code returned by acquisition function = acq

- **Acq fr nb** → Number of frames per acquisition

- **BW [MB/s]** → Evaluation of data rate produced by DAQ

Click on “Start” button to start DAQ emulation, on “Stop” to stop it and on “Print parameters” to print DAQ emulation parameters record value in log window & file.
Check frames content subpanel

This panel shows on-line the values of Mimosa 26 data stream ‘relevant fields’ of the frame selected by the control ‘Frame No’

- **Header** of each Mimosa 26
- **Trailer** of each Mimosa 26
- **Frame counter** of each Mimosa 26
- **Data length** [in bytes] of each Mimosa 26
- **The triggers number**
- **The first three triggers** (TLU) + last one
- **The first three time stamps** (Flex RIO) + last one

It’s also possible to display frames off-line when emulation has been stopped. The eudet_frio lib keep in a buffer all the frames of current acquisition. Therefore it’s possible to scan them off-line, specify the index of the frame in field ‘Frame No’, it will display content.

If you want to display frames content in text mode, select a print level via the control ‘No print’, move between frames and look in the errors and messages Window.
A verification of all frames of each acquisition can be done on-line. You can select the check level via the control “Check mode”, errors are count in the display “Errors cnt”, which you can reset via button “Reset→”.

It’s also possible to display frames loaded from a file. Load a file via the sub panel “Load run”, the “Acq No” control will be enabled and allows you to select the acquisition to scan via “Frame No” control. You must select the “Acq No” first and after you can display frames by selecting them via “Frame No”. If you forget to specify “Acq No” bad results may be displayed.
5 Procedure to start emulation

Initialize the software, need to be done only one time at beginning.

Set run configuration, the Main parameters are the Mimosa 26 number and the data transfer mode. Then click on “Conf Run”.
Set emulation configuration → header, trailer, triggers ... Start emulation by a click on "Start" button.

Look at results in “Check frames content” sub panel.

You can stop emulation via “Stop” button and go to “Run control” panel to select others run parameters.
6 Playing with the DAQ emulator

6.1 Mode EUDET 1

6.1.1 Fixed frame size

Emulator in mode **EUDET 1**, **Mimosa 26**, default frame size, **three trigger** are generated and **frame No 0** is displayed.
Now frame No 5 is displayed
We print header of frame No 5 in log windows

This is the print result
We print header + triggers of frame No 5 in log windows

This is the print result
The frame is also printed in the log file `x:\log\msg_emul_flex_rdo_daq.txt`.
6.1.2 Maximum frame size

Emulator in mode EUDET 1, 6 Mimosa 26, maximal frame size, three trigger are generated and frame No 0 is displayed.

The first Mimosa 26 has maximum data length (2280 bytes), others keep default data length.
6.1.3 Random frame size

Emulator in mode EUDET 1, Mimosa 26, random frame size, three trigger are generated, and frame No 0 is displayed.

The six Mimosa 26 have a random data length.
6.2 Mode EUDET 2

6.2.1 Default frame size & no trigger

Emulator in mode EUDET 2, 6 Mimosa 36, default frame size, no trigger and frame No 0 is displayed.

If there is no trigger “ F0000 – T0000 ” is displayed.
6.2.2 Default frame size & one trigger

Emulator in mode EUDET 2, 6 Mimosa 26, default frame size, one trigger generated and frame No 0 is displayed.

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel.
6.2.3 Default frame size & five triggers

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel. The GUI displayed the first three triggers plus the last one of the five.
6.2.4 Default frame size & ten triggers

Emulator in mode EUDET 2, 6 Mimosa 26, default frame size, ten triggers displayed.

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel.

Printing of frame header and trigger list is selected.
### 6.2.5 Print result.

We can see the trigger list, first three and last one are set with the values configured in GUI, others are set to 0.
6.3 Mode EUDET 3

6.3.1 Default frame size & no trigger

Emulator in mode EUDET 3, 6 Mimosa 26, default frame size, no trigger, frame No 0 has a strange pattern and bandwidth field indicates 0!

In EUDET mode 3, only frames with trigger are read. Therefore as there is no trigger, there are no frames, a default pattern is displayed for frame 0 and bandwidth is 0 because there is no data transfer.
6.3.2 Default frame size & one trigger

Emulator in mode **EUDET 3, 6 Mimosa 26**, **default frame size**, **one trigger generated** and frame No 0 is displayed.

Now there is **one trigger**, **bandwidth is not zero** and frame No is displayed.

The **triggers and time stamp** values displayed in the **bottom panel** are the ones configured as emulation parameters in the top panel.
6.3.3 Maximum frame size & five triggers

Emulator in mode EUDET 3, 6 Mimosa 26, maximum frame size, five triggers generated and frame No 0 is displayed.

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel.
6.3.4 Maximum frame size & ten triggers

Emulator in mode EUDET 3, 6 Mimosa 26, maximum frame size, ten triggers generated and frame No 0 is displayed.

Now there is one trigger, bandwidth is not zero and frame No is displayed.

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel.

Printing of frame header and trigger list is selected.
6.3.5 Print result.

We can see the trigger list, first three and last one are set with the values configured in GUI, others are set to 0.
6.3.6 Maximum frame size & 1 trigger / 100 frames

Emulator in mode EUDET 3, Mimosa 26, maximum frame size, 1 trigger / 100 frames generated and frame No 0 is displayed.

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel.

Print minimal information ➔ acquisition and frame No.
6.3.7 Print result.

The frames acquired are 0,1,2,3 – 100,101,102,103 – 200, 201, 202, 203 etc ...

We configured emulator to generate one trigger each 100 frames, therefore we should get the frames 0 – 100 – 200 etc ... it’s the case ☺

But we also get three following frames, that’s because we have configured the DAQ to acquire also the three frames following the trigger. This is done by setting the constant EFRIO__FRAME_NB_TO_READ_AFTER_TRIG to 3 in eudet_frio.def file.
6.3.8 Maximum frame size & 3 consecutive triggers / 100 frames

Emulator in mode EUDET 3, 6 Mimosa 26, maximum frame size, 3 consecutive triggers / 100 frames generated and frame No 0 is displayed.

The triggers and time stamp values displayed in the bottom panel are the ones configured as emulation parameters in the top panel.

Print minimal information ➔ acquisition and frame No.
6.3.9 Print result.

The frames acquired are 0,1,2,3,4,5 – 100,101,102,103,104,105 – 200, 201, 202, 203,204,205 etc ...

We configured emulator to generate three triggers each 100 frames, therefore we should get the frames 0,1,2 – 100,101,102 – 200,201,202 etc ... it's the case ☺☺ ☺☺

But we also get three following frames, that's because we have configured the DAQ to acquire also the three frames following the trigger. This is done by setting the constant EFRIO__FRAME_NB_TO_READ_AFTER_TRIG to 3 in eudet_frio.def file.
6.3.10 Random frame size & 3 consecutive triggers / 100 frames & save to disk

Emulator in mode EUDET 3, 6 Mimosa 26, maximum frame size, 3 consecutive triggers / 100 frames generated and frame No 0 is displayed.

Saving data to run file RUN_666 in directory e:\data is enabled.
Run file RUN_666.bin created on disk in directory e:\data.
6.3.11 Load a run from disk

**Load the run file** created in 6.3.7 via the panel “Load Run”, if loading is successful the status field switch to green, otherwise it will get red.

The run parameters, of the loaded file, are displayed in “Run Control” panel.
You can scan frames in the run

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>00001000</td>
<td>00001003</td>
<td>00001003</td>
<td>00001004</td>
<td>00001005</td>
</tr>
<tr>
<td>Trace</td>
<td>AAAAA0009</td>
<td>AAAAA0001</td>
<td>AAAAA0002</td>
<td>AAAAA0003</td>
<td>AAAAA0005</td>
</tr>
<tr>
<td>Front</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dil length</td>
<td>540</td>
<td>1238</td>
<td>1016</td>
<td>2138</td>
<td>46</td>
</tr>
<tr>
<td>Target</td>
<td>P0000 - T0040</td>
<td>P0000 - 10000</td>
<td>P0000 - T0000</td>
<td>P0000 - T0000</td>
<td>P0000 - T0000</td>
</tr>
<tr>
<td>Time stamp</td>
<td>P0000 - L0000</td>
<td>P0000 - L0000</td>
<td>P0000 - L0000</td>
<td>P0000 - L0000</td>
<td>P0000 - L0000</td>
</tr>
<tr>
<td>AeqNo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Check node</td>
<td>0 No</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

first specify the acquisition No and after the frame No to display

We can also print frames in text mode

We get the same sequence of frames – 0,1,2,3,4,5 – 100,101,102,103,104,105 etc – as the one in 6.3.7 when we took the run
7 How to interface emulator to EUDET DAQ?

7.1 Introduction

I never work with EUDET DAQ software, and I never find the time to read all the documentation about it. Therefore I will propose a sketch of interfacing and we will adjust it “on-line” if it doesn’t fit well or is not applicable ...

We can imagine a sequence in four steps:

- EUDET DAQ send a request to emulator to configure run parameters
- EUDET DAQ send a request to start emulation
- EUDET DAQ wait for data from emulator
- EUDET DAQ send a stop request to stop emulation

The emulator may have more parameters than EUDET can provide, it would not be a problem → they can be hard coded on emulator side.

In the current version of the software the emulator get his parameters from GUI controls, copy them in global variables named “context records” and call eudet_frio library functions to execute actions.

We can add a remote control option, selected via check box on GUI, if it’s enabled all GUI controls no more act as controls but as indicators. They receive theirs values from EUDET DAQ via Ethernet. The GUI “Run Control” and ”Flex RIO board emulation” panels will be simple display of request send by EUDET DAQ.

I think that this approach is close to what we need for the real DAQ control.

On emulator software point of view we have two directions

- The input → run control & emulation request
- The output → telescope data stream
7.2 The input side ➔ Run control & Start emulation

7.2.1 Run control context record and configuration function

**Context** record EFRIO__TRunCont ➔ eudet_frio.typ

```c
typedef struct {
    uint16 Parameter1; // Run 26 number
    uint16 Parameter2; // Frame number per acquisition
    uint8 Parameter3;  // Run 26
    uint8 Parameter4;  // Total event number of run
    uint8 Parameter5;  // Event number per file
    char Parameter6[64]; // Run file destination directory
    char Parameter7[64]; // Prefix of run file name, eg "EVE_006" as the prefix
    char Parameter8[128]; // Transfer mode see e.g. EFRIO__TRUN in ».def file
    uint8 Parameter9; // Timesec mode ➔ Future use
    uint8 Parameter10; // Save data on disk
    uint8 Parameter11; // Send data on Ethernet
    uint8 Parameter12; // % of data sent on Ethernet
    uint8 Parameter13; // Enable data rate measurement, hard coded in EFRIO__FootRun [...] #
    uint8 Parameter14; // Key number used to measure data rate, hard coded in EFRIO__FootRun [...] #
    char Parameter15[64]; // Not used yet
    char Parameter16[64]; // Dim data ParameterTransferMode = EFRIO__FootRun [...] #
    char Parameter17[64]; // Size of acquisition frames buffer
    char Parameter18[64]; // Size of ParameterTransferMode = EFRIO__FootRun [...] #
    char Parameter19[64]; // Not used yet
    char Parameter20[64]; // Dim data ParameterTransferMode = EFRIO__FootRun [...] #
    char Parameter21[64]; // Not used yet
    char Parameter22[64]; // Not used yet
    char Parameter23[64]; // Variables to measure data rate ➔ average over ParameterTransferMode data acquisitions
    char Parameter24[64]; // Total size acquired during ParameterTransferMode data acquisitions
    char Parameter25[64]; // Start time of measurement
    char Parameter26[64]; // Stop time of measurement
    char Parameter27[64]; // Total time of measurement
    char Parameter28[64]; // Return code of any function
    char Parameter29[64]; // Acquisitions counter
    char Parameter30[64]; // Frames counter
    char Parameter31[64]; // Frames counter ➔ By default events counter ➔ frames counter
    char Parameter32[64]; // But they may be different as more than one frame is needed to build a physics event
    char Parameter33[64]; // Return code of any function
    char Parameter34[64]; // Buffer for frames
    // Only one of the two is allocated depending on ParameterTransferMode = EFRIO / EUDT
    char Parameter35[64]; // Buffer for frames
    // If data ParameterTransferMode = EFRIO ➔ acquisition frames buffer
    char Parameter36[64]; // Buffer for frames
    // If data ParameterTransferMode = EUDT ➔ acquisition frames buffer
    char Parameter37[64]; // Buffer for frames
} EFRIO__TRunCont;
```
**Configuration function** `EFRIO_FConfRun ( ... )` → `eudet_frio.c`

**Less parameters than fields on EFRIO__TRunCont ;-)**

```
/* ****************************************************************************
Prototype : int32_t EFRIO_FConfRun | int32_t MxExpB, int32_t ExpNo, int32_t ExpTime, int32_t
                      EFRIOFile, int32_t FrameNPerExp, int32_t DataTransferMode, char* PerfDir,
                      char* FileNamePrefix, int32_t SaveOnDisk, int32_t SendOnEth, int32_t SendOnETHPCent |
:                      :                      :                      :                      :                      :                      :
Goal : Configure parameters, eg : Get them from GUI or Ethernet
Inputs : MxExpB - MxExpB number in the Exp
         ExpNo - Exp no
         ExpTime - Exp time in run
         EFRIOFile - EFRIO file name
         FrameNPerExp - Frames number per acquisition
         DataTransferMode - Data transfer mode
         PerfDir - Performance directory
         FileNamePrefix - Prefix of run file name | eg : RUN_26_ | "RUN" in the prefix |
         SaveOnDisk - Save or not data to disk
         SendOnEth - Send or not data to Ethernet | if SendOnEth = 1 |
Outputs : The function returns
          : 0 if ok
          : -1 if an error occurs
          :                      :
Globals :                      :
Remark :                      :
Level :
Date : 06/05/2010
Rev :
Desc : Sub to disk
Author : Gilles CLAUD
E-mail : gilles.clau@in2p3.fr
License : IN2P3 */
```

- 57 -
Context printing function EFRIO_FPrintRunContRec ( ... )

eudet_frio_print.c

This function prints run context record in messages window and log file.

```c
Prototype : SIInt32 EFRIO_FPrintRunContRec ( EFRIO__TRunCont* PtRec )

Goal : Print run context record in log file

Inputs : PtRec - Pointer on the record

Outputs : The function returns

: 0 if ok
: -1 if PtRec = NULL

Globals :

Remark :

Level :

Date  : 09/30/2010
Doc date : 07/11/2010

Author : Gilles CLAUS

Email : gilles.claus@ires.in2p3.fr

Labo : IFRIC */
```

- 58 -
Part of code called by a click on button “Conf Run”

```c
// ------------------------
// Conf run
// ------------------------

// Call DLL run Configuration function with parameters get from GUI

VRet = KF810_FConfRun ( 
    VUniGb, 
    VRunNo, 
    VToeEthernet, 
    VToePortFile, 
    VToePortKey, 
    VConfTrigMode, 
    0 /* Trigger */), 
    VTimeString, 
    VSaveOnDisk, 
    VSendOnEth, 
    VSendOnFTPPost );

// Update status fields & enabled / disable some panel controls

if ( VRet == 0 ) { 
    GrpHunCtlDispStatus->Text = "Conf run done !";
    GrpHunCtlDispStatus->Color = c1Green;
    GrpHunCtrl->Enabled = True;
    GrpHunCtrl->Enabled = False;
} else { 
    GrpHunCtlDispStatus->Text = "Conf run failed !";
    GrpHunCtlDispStatus->Color = c1Red;
    GrpHunCtrl->Enabled = True;
    GrpHunCtrl->Enabled = False;
}
```

- 59 -
7.2.2 Emulation context record and configuration function

**Context record EFRIO__TAcqEmul → eudet_frio.typ**

```c
typedef struct {
    UInt32 ParAcqCycleEMS; // Delay between two acquisitions
    UInt32 ParEmuleDwmsSendEmu; // Delay added to PC RAM access to emulate flex RIO RAM access time
    UInt32 ParEmuleFuncNo; // Select emulation function to call - future use - not implemented now
    UInt32 ParEmuleFuncPar; // Enable random generation of data size per function 26
    // By default data size is fixed in emulation function
    // Used to check if variabel length records are properly handled
    UInt32 ParSetEmulateSoOneMsg; // Set maximum possible data set on first Mi6; overwrite value set by em function, but next Mi6 keep the data size value from emulation funct
    UInt32 ParHeader[EFRIO__MAX_ABIC_NR]; // Emulated header of each Mi6
    UInt32 ParTrailer[EFRIO__MAX_ABIC_NR]; // Emulated trailer of each Mi6
    UInt32 ParTriPMiheaderFrame; // Number of trigger per frame, set the part trigger nb (8192) of Mi6
    // In data transfer modes EUDET 2 4 3 a more complex trigger emulation is
    // We don’t emulate ParTriPMiPerFrame on each frame but on 8 consecutive
    // each N frames
    UInt32 ParTriPMiOneFrameOverMi; // Start emulate ParTriPMiPerFrame on one frame over M = ParTriPMiOneFrame
    UInt32 ParTriPMiConsecutiveParameters; // Emulates on 16 consecutive frames = ParTriPMiConsecutiveParameters
    // TLU trigger & Flex RIO trigger emulation
    // Up to 188 couples TLS & Flex RIO triggers can be emulated but only 16 are configurable, from OUT, one EFRIO__MAX_EMUL_OUT_TO_FLO_NR = 4
    // First three are configurable from OUT
    // The last one is configurable from OUT
    // Others are configured in emulation function and set to 0
    UInt32 ParTriPMi[EFRIO__MAX_EMUL_GUT_TRIG_NR]; // Emulated TLU trigger
    UInt32 ParTriPMi[EFRIO__MAX_EMUL_GUT_TRIG_NR]; // Emulated Flex RIO trigger, called “true step”
    // DVAR info to emulate flex RIO rendout (we need a PC RAM block of same
    // UInt32 ParTriDMems; // DVAR size in BU
    UInt32 ParTriDMems; // DVAR size in bytes
    UInt32 ParTriDMems; // DVAR pointer
    char ParTriDMems[OLS_CRT_S2]; // A constant set by emulation function selected by ParEmuleFuncNo
    // Future use - not implemented now
    // DVAR emulation results
    // UInt32 ParKcpCnt; // Acquisition counter
    UInt32 ParKcpCnt; // Acquisition counter
    UInt32 ParKcpCnt; // Events counter
    UInt32 ParKcpCnt; // Error code returned by acquisition function
} EFRIO__TAcqEmul;
```

- 60 -
Function which fill EFRIO__TAcqEmul with parameters from gui

WinMain.cpp

```c
void __fastcall TFormMain::FormBoardGetPar ( int Caller )
{
    SInt6 Vi;
    EFRIO__TAcqEmul* VPtAcqEmul = (EFRIO__VContext.AcqEmul);
    EFRIO__RunCont* VPtRunCont = (EFRIO__VContext.RunCont);
    EFRIO__BoardConf* VPtBoard = (EFRIO__VContext.ABoardsConf);

    // Get param from GUI
    VPtAcqEmul->ParAcqCycleMs = (SInt6)GrpBoardEdAcqCycleMs->Checked;
    VPtAcqEmul->ParAcqModeAms = (SInt6)GrpBoardEdAcqModeAms->Checked;
    VPtAcqEmul->ParAcqModeBms = (SInt6)GrpBoardEdAcqModeBms->Checked;
    VPtAcqEmul->ParAcqModeCms = (SInt6)GrpBoardEdAcqModeCms->Checked;
    VPtAcqEmul->ParAcqModeDms = (SInt6)GrpBoardEdAcqModeDms->Checked;
    VPtAcqEmul->ParAcqModeEms = (SInt6)GrpBoardEdAcqModeEms->Checked;
    VPtAcqEmul->ParAcqModeFms = (SInt6)GrpBoardEdAcqModeFms->Checked;
    VPtAcqEmul->ParAcqModeGms = (SInt6)GrpBoardEdAcqModeGms->Checked;
    VPtAcqEmul->ParAcqModeHms = (SInt6)GrpBoardEdAcqModeHms->Checked;
    VPtAcqEmul->ParAcqModeIms = (SInt6)GrpBoardEdAcqModeIms->Checked;
    VPtAcqEmul->ParAcqModeJms = (SInt6)GrpBoardEdAcqModeJms->Checked;
    VPtAcqEmul->ParAcqModeKms = (SInt6)GrpBoardEdAcqModeKms->Checked;
    VPtAcqEmul->ParAcqModeLms = (SInt6)GrpBoardEdAcqModeLms->Checked;
    VPtAcqEmul->ParAcqModeMms = (SInt6)GrpBoardEdAcqModeMms->Checked;
    VPtAcqEmul->ParAcqModeNms = (SInt6)GrpBoardEdAcqModeNms->Checked;
    VPtAcqEmul->ParAcqModeOms = (SInt6)GrpBoardEdAcqModeOms->Checked;
    VPtAcqEmul->ParAcqModePms = (SInt6)GrpBoardEdAcqModePms->Checked;
    VPtAcqEmul->ParAcqModeQms = (SInt6)GrpBoardEdAcqModeQms->Checked;
    VPtAcqEmul->ParAcqModeRms = (SInt6)GrpBoardEdAcqModeRms->Checked;
    VPtAcqEmul->ParAcqModeSm = (SInt6)GrpBoardEdAcqModeSm->Checked;
    VPtAcqEmul->ParAcqModeTms = (SInt6)GrpBoardEdAcqModeTms->Checked;
    VPtAcqEmul->ParAcqModeUms = (SInt6)GrpBoardEdAcqModeUms->Checked;
    VPtAcqEmul->ParAcqModeVms = (SInt6)GrpBoardEdAcqModeVms->Checked;
    VPtAcqEmul->ParAcqModeWms = (SInt6)GrpBoardEdAcqModeWms->Checked;
    VPtAcqEmul->ParAcqModeXms = (SInt6)GrpBoardEdAcqModeXms->Checked;
    VPtAcqEmul->ParAcqModeYms = (SInt6)GrpBoardEdAcqModeYms->Checked;
    VPtAcqEmul->ParAcqModeZms = (SInt6)GrpBoardEdAcqModeZms->Checked;

    for (Vi = 0; Vi < EFRIO__MAX_ADC_HD; Vi++)
    {
        VPtAcqEmul->ParAcqHeader[Vi] = FDEditBoxToInt ( GrpBoardEdAcqHeader[Vi] );
        VPtAcqEmul->ParAcqHeader[Vi] = FDEditBoxToInt ( GrpBoardEdAcqHeader[Vi] );
    }

    for (Vi = 0; Vi < EFRIO__MAX_ADC_HD; Vi++)
    {
        VPtAcqEmul->ParAcqHeader[Vi] = FDEditBoxToInt ( GrpBoardEdAcqHeader[Vi] );
        VPtAcqEmul->ParAcqHeader[Vi] = FDEditBoxToInt ( GrpBoardEdAcqHeader[Vi] );
    }

    // WinMain.cpp source code continues...
```
**Context printing function** EFRIO_FPrintAcqEmulRec (...) → eudet_frio_print.c

This function prints run context record in messages window and log file.

```c
/* DOCUMENTATION */

Prototype : SInt32 EFRIO_FPrintAcqEmulRec ( EFRIO_TAcqEmulRec PtRec ) ;

Goal : Print acquisition emulation context record in log file

Inputs : PtRec - Pointer on the record

Outputs : The function returns

0 if ok
-1 if PtRec = NULL

Globals :

Remark :

Level :

Date : 31/10/2010
Doc date : 07/11/2010
Author : Gilles CLAUS
E-mail : gilles.claus@ires.in2p3.fr
Labo : IPHC */
```

`eudet_frio_print.c`

This function prints run context record in messages window and log file.
Part of code called by a click on button “Start”

```c
void __fastcall TForm1::GrpEmuiBoardDlgStartClick(TObject *Sender)
{
  int32 VRet = 0; // Variable to store error code of functions called

  EFRIO_TAskEnv1* VPtrAcqEnv1 = EFRIO_VGContext.AcqEnv1; // Pointer to acq e 
  // -------------------------------
  // Init DAQ emulation
  // -------------------------------

  EFRIO__FDInitBegin ( 0 /* RunInLibview */ );
  // -------------------------------
  // Get parameter from GUI
  // -------------------------------

  GrpEmuiBoardSetFam ( 0 );
  // -------------------------------
  // Display info
  // -------------------------------

  FReadInt2Edit ( VPtrAcqEnv1->InfoEmuStrND, GrpEmuiBoardDispDispSamStrR );

  GrpEmuiBoardDispExtraChan->Checked = (bool) VPtrAcqEnv1->InfoExtraChan;
  // ----------------------------------------------------------------------
  // If saving is enabled ( save pzt ) => create run conf/pzt a data files
  // ----------------------------------------------------------------------

  EFRIO__FStartSavingOnFile ();
  // -------------------------------
  // Start acq emulation timer
  // -------------------------------

  TEmuiAcqCycle->Interval = VPtrAcqEnv1->FTimeCycleMs;
  TEmuiAcqCycle->XEnabled = True;
}
```
7.2.3 How to access to context records \( \Rightarrow \) which variables?

If the code is written in eudet_frio library we can access via the global variable `EFRIO__VGContext` which contains all variables of library.

```c
typedef struct {
    SInt8 InitInitCont; // Lib init done or not
    EFRIO__TRunCont *RunCont; // Run context = parameters, memory
    EFRIO__TAcqEmul *AcqEmul; // Acq emulation context
    EFRIO__TCheck FcCheck; // Frame check functions context
    EFRIO__TRunCont RunCont; // Run context = parameters, memory
    EFRIO__TFrameList AcqFrameList[1]; // Frame list of acquisition - Can be
    // List of frame Id to read ( EudetNode $$\Rightarrow$$ Trigger + 2 following frames ) / acquisition - Can
    SInt16 AcqFrameWithTrigList[1][EFRIO__MAX_FRAME_NR_PER_ACQ];
    EFRIO__TTriggerRec* PctmpTriggerRec; // Temporary triggers record used for
} EFRIO__TContext;
```

You can use the following fields:

- **RunCont** to access run context record \( \Rightarrow \) **EFRIO__TRunCont**
- **AcqEmul** to access acquisition emulation context \( \Rightarrow \) **EFRIO__TAcqEmul**

If the code is written outside eudet_frio library \( \ldots \) you will find yourself the way \( \ldots \) If you are afraid about global variables malediction \( \ldots \) you can write a function which return a pointer to `EFRIO_VGContext` ;-) or encapsulate it in a class with a method to access to each field \( \ldots \)
7.3 The output side ➔ Telescope data stream

7.3.1 How the Flex RIO board is read?

The flex RIO board acquires bunches of consecutive frames, then the software read the board. One of this bunches is called “an acquisition”, the default number of consecutive frames stored in one acquisition is 1800 (for historical reasons). As long as the software “runs fast enough” there is no missing frames from one acquisition to the next one. The period between acquisitions is ~ 207 ms for 1800 frames / acquisition.

As the time between two acquisitions is ~ 207 ms, we can use a timer to call the board readout function. It’s done like this, with a timer in DAQ emulator, in Labview DAQ application it’s done with an endless loop because we want to minimize the risk to loose frames.

![Image of software interface]

This is a part of the timer callback function which call the eudet_frio library functions which process data (frame with trigger selection, etc...)

➔ EFRIO__MI26_FFRioAcqDeserDataMi26 ( ... )

```c
// Call Flex RIO "acq deser data function"
// ---------------------------------------

VFtAngEvent->RunAngEventResetCode = EFRIO__MI26_FFRioAcqDeserDataMi26();

VFtNumCost->ParMi26Set, 0 /* BoardId */,
VFtNumPlo0Rwam, 0 /* FI02cMi26AInt */,
(VFtAngEvent->IntFi2RmemSc) / 4 /* Fl11w */,
0 /* AcqStatus */,
0 /* AcqStatus */,
0 /* WaitMskEnd */,
VFtNumCost->ParDataTransferMode, 0 /* TriggerHandlingMode */,
VFtModeCode);
```

- 65 -
The EFRIO__MI26_FFRioAcqDeserDataMi26 (...) :

- Get access to Flex RIO data via parameter PtSrcW32AsPt or PtSrcW32AsInt
- Call the emulation function if needed
- Call the add-hoc function to process data depending of “DataTransferMode” and Mimosa 26 number.

This is the function comment header in eudet_frio.c file.
Prototype: ETPRO_Host_FPOC_EUDET_FPeripheralGetAdresAI26 (  
    SINIE Interface, SINIE BoardId, SINIE* PRecRIOAddr, UINT16* PRecRIOAddr,  
    SINIE* EIO2, SINIE acquisition, SINIE trigger, SINIE WaitBeforeEnd,  
    SINIE* FuzzyConvertNode, SINIE TriggerHandlingNode, SINIE EmulateNode)

Goal:
This function is the upper level of Flex RIO readout functions, it calls
the right readout function depending on EIO2 & FuzzyConvertNode parameters.
On Labview side, this function is encapsulated in a VI of the same name,
which is called each time an acquisition is finished.

This function also call the frames emulation functions if emulation mode
is enabled.

Inputs:
   NI2000  - Number of Nima 36 to acquire
   BoardId  - Board identifier
   *PRecRIOAddr - Pointer on Flex RIO DRAM as pointer
   *PRecRIOAddrInc - Pointer on Flex RIO DRAM as an integer
   EIO2  - Size of Flex RIO DRAM in U16 ( 1 Unit = 1 U16 )
   AcqStatus  - Acquisition status flag provide by board
   Trigger  - Trigger status flag provide by board
   WaitBeforeEnd - Wait an end of function to measure free time
   *FuzzyConvertNode - *FuzzyTransferNode of ETPRO_FConvFunc
   *TriggerHandlingNode for data information
   *EmulateNode - Mode of trigger operation

    EnableNode - Enable frames emulation mode
    0 -> No frames emulation
    1 -> Emulation mode
    Mode NI26 < ENDET  1 -> 3 triggers / frame
    Mode ENDET 2 < 4 -> 1 trigger / frame
    < 0 -> Emulation mode
    Mode NI26 < ENDET  1 -> 3 triggers / frame
    Mode ENDET 2 < 4 -> | EnableNode | triggers / frame

Outputs:
The function returns
  0 if ok
  -1 if an error occurs

Globals:

Remark:
This is the part of EFRIO_M26_FFRioAcqDeserDataMi26 (...) which calls the add-hoc data processing function in mode EUDET 3.

```c
// EUDET node 3
if | DataConvertMode == EFRIO__TFP_NODE_EUDET_3__TWO_CHAN_SEND_FRAMES_WITH_TRIG | |
switch | Mi20Mb |
  case 1 : |
    Vm1 = EFRIO__MI26_FFRioAcqDeserDataEUDET3NodeMi26 (BoardId, PtIoMi26xPt, ItmRh, AmpStatus, TrigStatus, DataConvertMode | |
      break |
  case 6 : |
    Vm1 = EFRIO__MI26_FFRioAcqDeserDataEUDET3NodeMi26 (BoardId, PtIoMi26xPt, ItmRh, AmpStatus, TrigStatus, DataConvertMode | |
      break |
  default |
    | exit_redial ( -1, [ERR_OUT,"short : EFRIO__TFP_NODE_EUDET_3__TWO_CHAN_SEND_FRAMES_WITH_TRIG -> This number of Mi16 * id is no |
      break |
    |
break |
```
7.3.2 How the data stream is organized?

The data stream is organized "per acquisition", if the number of frames per acquisition is set to 1800, the period between two acquisition will be 1800 x 115.2 µs (Mimosa 26 frame duration) = ~ 207 ms. It means that each 207 ms an acquisition will be ready in library memory, the number of frames will depend of the number of triggers therefore it can be less than 1800.

The eudet_frio library allocates a buffer large enough to contain an acquisition of the specified frames nb per acquisition value set in run control. This will be a single bloc of RAM on which you can have access in two ways:

- Pointer on the bloc beginning ➔ PtFrame
- An array of pointers on each frame ➔ AFramePtr[i]

It means that you can send on Ethernet the whole acquisition defined by PtFrame and it’s size or frame by frame by scanning the array AFramePtr[i]. It’s up to you to decide which method is the best.
7.3.3 Organization of one frame \(\rightarrow\) EFRIO__TFrame

The type which define the frame is EFRIO__TFrame \(\rightarrow\) eudet_frio.typ

```c
/* Definition of EFRIO__TFrame */
typedef struct {
    int32 Tag;            // EFRIO__FRAME_TAG
    int32 TotalSize;      // Total size of this frame
    int32 TriggerOffset;  // Offset (in bytes) from beginning of frame to trigger info part
    EFRIO__TFrameHeader Header; // Frame header
    EFRIO__TFrameData Data;  // Beginning of data part
}
EFRIO__TFrame;
```

The frame will contains the following items:

- **Size handling fields**
- A **header of fixed size** \(\rightarrow\) Header
- A **data part of variable size** \(\rightarrow\) Data
- After the data part a **list of triggers of variable size**
The first field "Tag" is used to tag beginning of frame in data stream. It can be helpful if someone need to deal with binary data. Its value is set by constant EFRIO_FLAME_TAG define in eudet_frio.h. This field can be removed by conditional compilation.

The second field "ToSz" indicates the total size of the current frame. If you want to go to next frames, set a byte pointer on current one, add "ToSz", cast the byte pointer to EFRIO_TFrame*, and it's done.

The third field "TrigRecOffset" indicates the position of the trigger record in the frame, it follows data part which has a variable size. To go to trigger record, set a byte pointer on beginning of frame, add "TrigRecOffset", cast the byte pointer to EFRIO_TTriggerRec*, and it's done.

The fourth field "Header" is the frame header, it has a fixed size.

The fifth field "Data" is the beginning of data part.

There is no field in EFRIO_TFrame for the triggers record because it's not possible as the Data field has a variable length, that's why we need the field "TrigRecOffset".
7.3.4 The frame header record \( \rightarrow \text{EFRIO\_TFrameHeader} \)

```c
*/
*/ Frame header
*/ ---------------------------------------------*/
*/ Each frame starts with a header which contains */
*/ - 192 bytes system info */
*/ - Mimosa 26 relevant fields */
*/ ---------------------------------------------*/
*/ Date: 22/10/2010 */
*/ Doc date: 04/11/2010 */
*/ Author: Gilles CLAUD */
*/ E-mail: gilles.clus@ipsa.in2p3.fr */
*/ License: DRS - IPHC */
*/ ---------------------------------------------*/

typedef struct {
    #define EFRIO__FRAME_TAG_HEADER  // EFRIO__FRAME_TAG_HEADER
    UInt32 Tag; // EFRIO__FRAME_TAG_HEADER
    #endif
    UInt16 AcqId; // Index of acquisition containing this frame
    UInt16 FrameIdInAcq; // Index of frame ID in the CURRENT acquisition
    UInt16 MapsName; // MAPS name as a 16 bits code
    UInt16 MapsNb; // Total number of MAPS in data
    UInt32 AMapsHeader[EFRIO_MAX_MAPS_NB]; // Mimosa 26 header field
    UInt32 AMapsFrameCen[EFRIO_MAX_MAPS_NB]; // Mimosa 26 frame center field
    UInt16 AMapsDeltaLen[EFRIO_MAX_MAPS_NB]; // Mimosa 26 data length in BYTES -> it's final
    UInt32 AMapsTrailer[EFRIO_MAX_MAPS_NB]; // Mimosa 26 trailer field
    UInt16 TriggerNb; // Total triggers number during this frame
    UInt16 AMapsTrigInfo[EFRIO_MAX_TRIGGER_NB_STORED_IN_FRAME_DATA]; // First 3 "26 triggers"
        // if more than 4 trigger
} EFRIO__TFrameHeader;
```

The first field “ Tag ” acts like EFRIO\_TFrame Tag, it is set to EFRIO\_FRAME\_TAG\_HEADER

**AcqId** and **FrameIdInAcq** indicates the index of the acquisition which contains this frame and the index of the frame (0..1799) in this acquisition.

**MapsName** is a code to identify the MAPS, **MapsNb** the number of MAPS in DAQ.

The fields **AMaps**... are arrays containing Mimosa 26 frame header, ...

**TriggerNb** contains the number of trigger during the current frame

**AMapsTrigInfo** contains the first three triggers keep for compatibility with our previous DAQ \( \rightarrow \) not useful for EUDET
7.3.5 The data part

```c
typedef struct {
    #ifdef EFRIO__FRAME_TAGS_ENABLE
    UInt32 Tag; // EFRIO__FRAME_TAG_DATA
    #endif
    UInt32 TotSz; // Total size of data bloc
    UInt32 OneMapSz; // Size of data of one MAPS
    UInt32 ADataW32[0]; // Beginning of data space
} EFRIO__TFrameData;
```

The first field "Tag" acts like EFRIO_TFrame Tag, it is set to EFRIO__FRAME_TAG_DATA.

The second field "TotSz" indicates the total size of data bloc.

The third field "OneMapSz" indicates the data size for one MAPS, must be multiplied by the number of MAPS to get the size [in bytes] of ADataW32 array.

The fourth field "ADataW32" is a pointer to the first W32 of data.
Flex RIO DAQ proposal : Mimosa 26 data stream

Readout configuration **N° 3** : 2 serial outputs @ 80 MHz

- Provides the whole states memory size : **1140 W16** (word of 16 bits) – **570 W16** / link

Summary

- Data generated on rising edge of Mimosa 26 clock
- Header → 16 bits / output
- Frame counter → 16 bits / output
- Data length (useful part of data) → 16 bits / output (Sum the 2 W16 to get matrix W16 size)
- Data → Max = 5 x 16 bits / output
- Trailer → 16 bits / output
- Padding zero → 32 bits / output
- Maximum stream size per output : **9216 bits** = **576 W16** = **1152 W8** ... Can be less → Defined by Data length field

14/12/2010
WARNING!

The Mimosa 26 data stream is multiplexed on two data links D00 and D01, as explained on previous page. The Flex Rio firmware has a 16 bits deserializer connected to each data link and it doesn’t demultiplex data after deserialization. Therefore this multiplexed data structure is still present in the “ADataW32” array of the “EFRIO__TFrameData”.

The data stream has been demultiplexed to fill the EFRIO__TFrameHeader fields (header, frame counter, data length trailer) but not for the data part. This data demultiplexing has not been implemented because this processing cost execution time and when the first version of code has been written I didn’t know if it would be better to make this processing on NI CPU side or on EUDET DAQ side.

As we have 100 ms free CPU time, it can be implemented on NI CPU side, but it is not done yet, the function EFRIO__Mi26_FFRioAcqDeserDataEudet3Mode6Mi26 must be modified to implement it.

Organization of data part in case of one Mimosa 26 is read by DAQ:

Wn = Word of 32 bits from array “ADataW32”

Data n link d0 = Word number n of 16 bits on data link d0

Data n link d1 = Word number n of 16 bits on data link d1

\[
\begin{align*}
W_0 &= D_{31:16} \\
W_1 &= D_{15:0} \\
W_2 &= D_{31:16} \\
&\quad \vdots \\
W_n &= D_{31:16} \\
\end{align*}
\]
**Organization of data part in case of six Mimosa 26 are read by DAQ:**

\[
W_n = \text{Word of 32 bits from array "ADataW32"}
\]

\[
\text{Data n link d0 chip } x = \text{Word number n of 16 bits on data link d0 of chip N}^x
\]

\[
\text{Data n link d1 chip } x = \text{Word number n of 16 bits on data link d1 of chip N}^x
\]

<table>
<thead>
<tr>
<th>(W_n)</th>
<th>D31D16</th>
<th>D15D00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data 0 link d1 chip 0</td>
<td>Data 0 link d0 chip 0</td>
</tr>
<tr>
<td>1</td>
<td>Data 0 link d1 chip 1</td>
<td>Data 1 link d0 chip 1</td>
</tr>
<tr>
<td>2</td>
<td>Data 0 link d1 chip 2</td>
<td>Data 2 link d0 chip 2</td>
</tr>
<tr>
<td>3</td>
<td>Data 0 link d1 chip 3</td>
<td>Data 2 link d0 chip 3</td>
</tr>
<tr>
<td>4</td>
<td>Data 0 link d1 chip 4</td>
<td>Data 2 link d0 chip 4</td>
</tr>
<tr>
<td>5</td>
<td>Data 0 link d1 chip 5</td>
<td>Data 2 link d0 chip 5</td>
</tr>
<tr>
<td>6</td>
<td>Data 1 link d1 chip 0</td>
<td>Data 1 link d0 chip 0</td>
</tr>
<tr>
<td>7</td>
<td>Data 1 link d1 chip 1</td>
<td>Data 1 link d0 chip 1</td>
</tr>
<tr>
<td>8</td>
<td>Data 1 link d1 chip 2</td>
<td>Data 1 link d0 chip 2</td>
</tr>
<tr>
<td>9</td>
<td>Data 1 link d1 chip 3</td>
<td>Data 1 link d0 chip 3</td>
</tr>
<tr>
<td>10</td>
<td>Data 1 link d1 chip 4</td>
<td>Data 1 link d0 chip 4</td>
</tr>
<tr>
<td>11</td>
<td>Data 1 link d1 chip 5</td>
<td>Data 1 link d0 chip 5</td>
</tr>
</tbody>
</table>

.....

.....

\[
W_n = \text{Data n link d1 chip 5 Data n link d0 chip 5}
\]
7.3.6 The trigger record → EFRIO_TFrameData

The first field “Tag” acts like EFRIO_TFrame Tag, it is set to EFRIO__FRAME_TAG_TRIG

The second field “TotSz” indicates the total size of trigger record

The third field “TrigNb” indicates the number of triggers

The fourth field “TrigType” indicates the type of trigger → reserved for future use

The fifth field “ATrig” is a pointer on triggers

```c
typedef struct {
  #ifdef EFRIO_FRAME_TAGS_ENABLE
  Uint32 Tag;    // EFRIO__FRAME_TAG_TRIG
  #endif
  Uint32 TotSz;  // Total size of trigger info bloc
  Uint16 TrigNb; // Total trigger nb
  Uint16 TrigType; // Type of trigger info stored
  Uint32 ATrig[0]; // Beginning off triggers list
} EFRIO__TTriggerRec;
```
7.3.7 The trigger record items

For each trigger eudet_frio library stores two triggers fields: first from TLU and second one from Flex RIO. It means that the array ATrig[] will contain TrigNb X 2 items. The TLU trigger info is written first, followed by the Flex RIO trigger/time stamp. It means that array organization will be as followed:

- Trigger [0] TLU $\rightarrow$ ATrig[0]
- Trigger [0] Flex RIO $\rightarrow$ ATrig[1]
- Trigger [1] Flex RIO $\rightarrow$ ATrig[3]
- ...
- ...
- ...
- Trigger [TrigNb-1] TLU $\rightarrow$ ATrig[(TrigNb X 2)]
- Trigger [TrigNb-1] Flex RIO $\rightarrow$ ATrig[((TrigNb X 2) + 1)]

TLU trigger record $\rightarrow$ EFRIO__TTluTrigger -= W32

```c
typedef union {
  UInt32 U32;
struct {
    UInt32 TrigInc : 14; // Trigger counter read from TLU
    UInt32 FrameStartIdx : 11; // Index of frame in current acquisition show
    UInt32 EventTakenByUser : 1; // For future use : Flag at 1 if BEN has take
    UInt32 Reserved : 3;
    UInt32 InvalidInfo : 1; // If 1 this field is not valid
  };
} EFRIO__TTluTrigger;
```
Flex RIO trigger / time stamp record \( \rightarrow \) EFRIO__TFlexRioTimeStamp1 = W32

```c
typedef union {
    UInt32 W32;
    struct {
        UInt32 Mi26Line : 10; // line of Mi26 read during which t
        UInt32 Mi26Frame : 21; // frame of Mi26 | = frame counter
        UInt32 InvalidInfo : 1; // If 1 this field is not valid
    } T;
}
EFRIO__TFlexRioTimeStamp1;
```
7.3.8 How to access to frames data → which variables?

If the code is written in eudet_frio library we can access via the global variable EFRIO__VGContext which contains all variables of library.

```c
typedef struct {
    Int32 InitState;           // Lib init done or not
    EFRIO__TProcConf AProcConf[EFRIO__MAX_BOARD_ID]; // Acquisition boards config
    EFRIO__TBoardStatus ABoardStatus[EFRIO__MAX_BOARD_ID]; // Acquisition boards status
    Int16 AProcEmul QtyEmul;   // B&Q emulation context
    EFRIO__TFcCheck FcCheck;    // Frames check functions context
    EFRIO__TRunCont RunCont;    // Run context = parameters, memory access
    EFRIO__TFrameList AFrameList[1]; // Frame list of acquisition - Can be
    EFRIO__TTriggerRec* FtmpTriggerRec; // Temporary triggers record used for
} EFRIO__TContext;
```

You can use the following fields:

- **RunCont.PtFrame → EFRIO__TFrame** → Access to full block
- **AAcqFrameList[0]. AFramePtr[FrameIndex]** → Access frame by frame
7.4 How / where to write the code ?

7.4.1 The eudet_frio library and DLL

It can be written in the eudet_frio library which is compiled as a DLL. The code can be C or C++. But for the interface to Labview, as far as I know, it must be simple C function, there is no easy way to interface a class to Labview. May be by encapsulation in ActiveX or .NET object ? I believe we don’t such “ funny things ”, please use C and if a class is needed make a kind of wrapper via some C functions call. We want performances and reliability, we don’t need state of the art in software development.

7.4.2 Run control context record and configuration function

A set of files had been set in eudet_frio lib for user code implementation

- Eudet_frio_usr.def ➔ Macros and constants
- Eudet_frio_usr.typ ➔ Types and classes definition
- Eudet_frio_usr.var ➔ Global variables
- Eudet_frio_usr.h ➔ Functions header
- Eudet_frio_usr.c ➔ C or C++ code

They are empty, fill free to use them for your own source code. Therefore we can easily provide a library upgrade without impacting your own source code.
7.5 Warning about files library

This library handles files I/O. It implements classes TCBinFile and TCStreamFile used by EUDET Flex RIO library (eudet_frio).

This library is in directory x:\lib\com\files

Warning about TCStreamFile class!

This class speed up disk access by

- Making direct disk access = non buffered
- Having it’s own thread to write data to disk, therefore saving is always done in background, it’s not stopped while board is busy.

But this class had been quickly designed to test the Flex RIO system hardware, therefore it has limitations and it had not been intensively tested. For example it creates a single file, the run is no split in different files … Therefore, if you decide to use it please do it carefully, test your code, report us bugs if needed.

Acknowledgement

This work is supported by the Commission of the European Communities under the 6th Framework Programme “Structuring the European Research Area”, contract number RII3-026126.