A scalable, portable DAQ system design

Martin L. Purschke

About me

Studied nuclear physics at the University of Muenster, Germany

WA80 - WA93 – WA98 Experiments at CERN

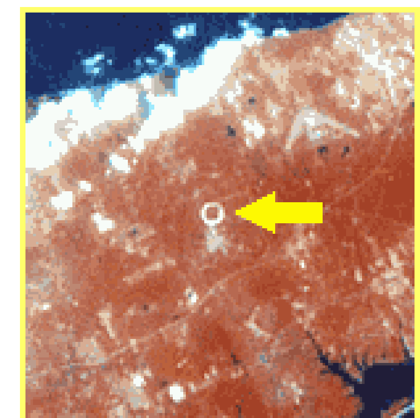
Graduated in 1990

Spent 11 years at CERN with the SPS Heavy-Ion Program until it ended in 1996

Moved to Brookhaven National Laboratory to work with the Relativistic Heavy Ion Collider

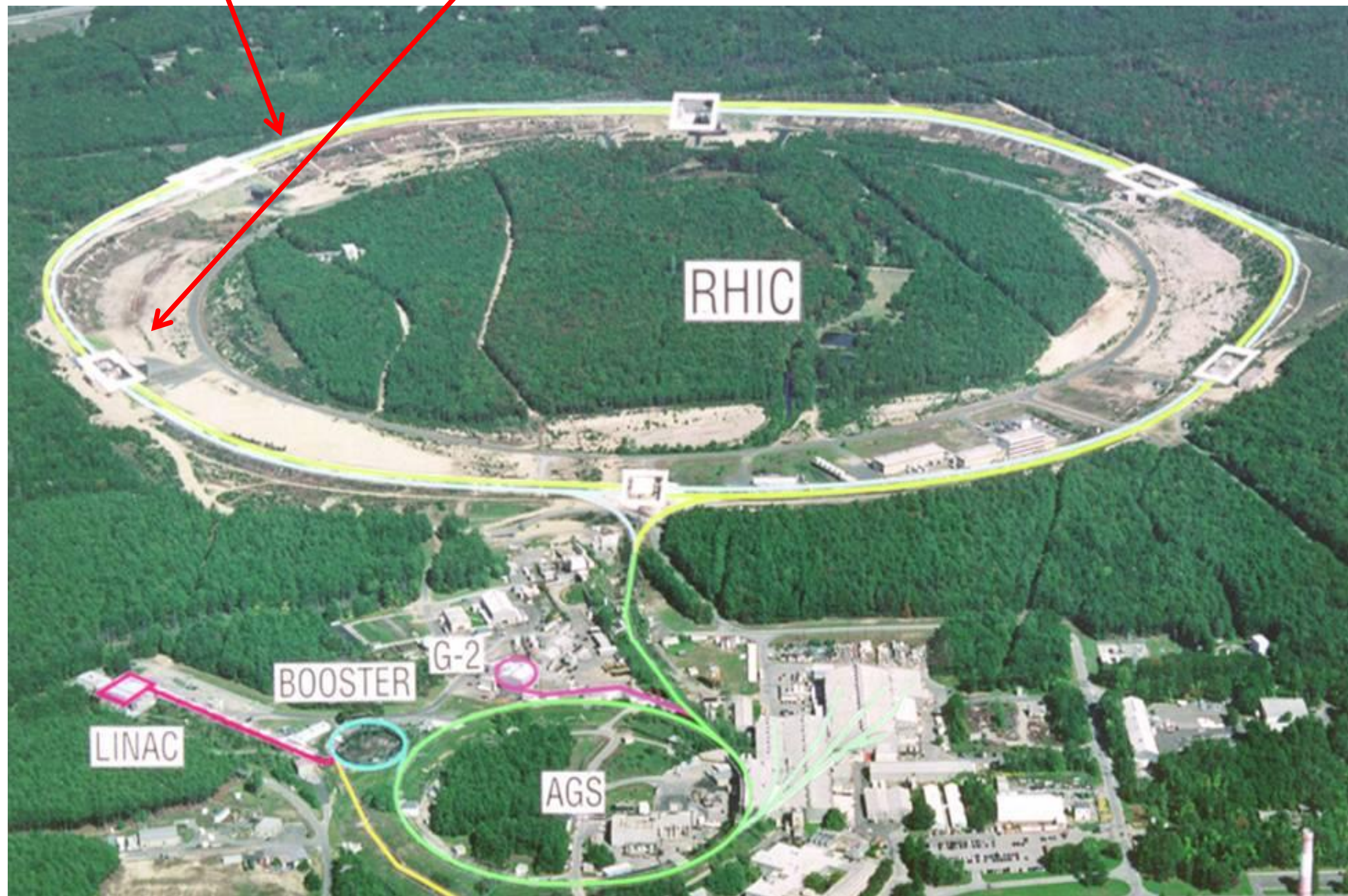
Am the DAQ coordinator for the PHENIX and the successor sPHENIX Experiments

Long Island, NY



RHIC from space

RHIC and PHENIX



The Relativistic ***Heavy Ion*** Collider

Huge variety of ions possible - Au, Cu, ^{3}He so far, but pretty much anything is possible

Polarized protons – a unique facility

500GeV/proton -> 200GeV/N for Au

Dedicated HI and p facility

PHENIX – high-rate heavy-ion experiment

Electromagnetic probes - Photons, electrons, muons

Heavy quarks & quarkonia

High Data Rates PHENIX

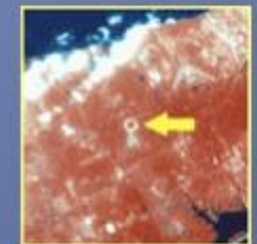
- At the LHC, the vast majority of collisions are truly “boring”
- So a trigger can be truly selective
- For Heavy Ions, virtually each event has some interesting feature
- Others you can’t trigger on easily because of the high multiplicity
- We have traditionally written the highest data rates for more than a decade

1.5GByte/s fully compressed data stream

LHC-Era Data Rates in 2004 and 2005 Experiences of the PHENIX Experiment with a PetaByte of Data

Martin L. Purschke, Brookhaven National Laboratory
PHENIX Collaboration

RHIC from space



Long Island, NY

1

My opening slide from the CHEP in 2006

What I'll be talking about

Over the course of your career, you likely see quite a number of different data acquisition systems

Some will be flexible, easy to use, adaptable, fun, others…. not so

I will go through a number of design principles that have served me well

If you ever get to design one, or (more likely) aid in the selection of an existing one, I hope this will be useful

(Obviously, my principles will be biased. Do I think my DAQ systems are great? You bet ☺).

I also don't believe in "Do as I say, not as I do…" – I will show you with the example of a particular DAQ system what I mean and how I implemented things

I will show you a number of examples to make this more tangible

Design Goals, also known as Buzzwords

- Modularity
- Data integrity, robustness and resilience
- No exposure of analysis code to internals
- Binary payload format agnostic
- No preferred endianess
- Support for data compression
- Different event types
- Set of tools to inspect / display / manipulate files
- Online monitoring support
- Electronic Logbook support
- OS integration
- Interface to community analysis tools (these days: root and 3rd-party frameworks)

That's quite a list. Let's go through and see what all that means

Data Formats in general…

One of the trickiest parts when developing a new application is defining a data format

It can take up easily half of the overall effort – think of Microsoft dreaming up the format to store this very PowerPoint presentation you are in a file. We used to have ppt, now we have pptx – mostly due to limitations in the original format design

A good data format takes design skills, experience, but also the test of time

The tested format usually comes with an already existing toolset to deal with data in the format, and examples – nothing is better than a working example

Case in point: Parts of the PHENIX Raw Data Format (PRDF) have their roots at the CERN-SPS, and the Bevalac Plastic Ball experiment in the 80's – that's a solid "test of time"

Resilience and error recovery

Imagine a data format where one bit error, or one error in some length field, in the data renders the entire file unreadable

Obviously not a good design – you will have such errors, corrupt tapes, recovered disk files, and you cannot allow to lose a significant portion of your statistics

Corrupt data is far more common than you think!

Data can be corrupted by the storage medium

Data structures can also be corrupt from the get-go by some bug in the DAQ

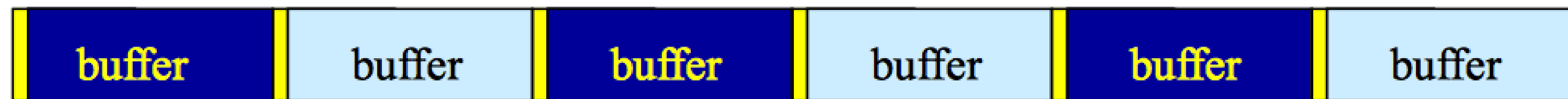
“Resilience in depth” – any corrupt entity must be able to be skipped, the remainder of the data recovered

You must also be able to account for what was lost

“*You must be able to erroneously feed your mail file to your analysis. It shouldn’t find events, but it shouldn’t crash, either.*”

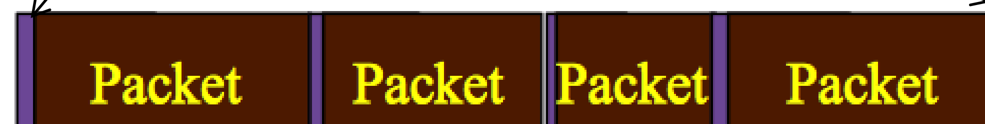
How did we implement this?

This is a storage-level layer, usually invisible



A variable number of Events per buffer

Data structures from individual detectors



The error recovery works on the smallest corrupted entity, a packet, an event, or a buffer.

Error Recovery

A good amount of the physical storage concept is derived from what was the main storage medium back in the 80's and 90's – tapes

Of course, in 2016 , we still write the majority of our data to tapes

Useful leftovers from the days of direct tape reading:

Our Buffers are a multiple of 8Kb "records" – tape drives used to write physical chunks of 8Kb

Got a corrupt data? Skip 8Kb records until you find the start of a new buffer. It must start on a record (8Kb) boundary. Without that constraint, you have no chance to find that.

Inside buffers, parts of the data of an event can be corrupt but the "outer" structure intact – skip event

Inside an event, the data structure from a detector can be corrupt – skip this and take a (user) decision whether or not to accept the event

At any time, you are in charge of dealing with the situation in a manner that suits your analysis.

No Preferred Endianess – what does that mean?

This is less of an issue today as it was 10 years ago when we had a lot of VME-based stuff and Motorola 68K and PowerPC CPUs in front-ends (all big-endian) and Intel/AMD for analysis (all little-endian)

Endianess – the order how a 2 or 4-byte variable is stored

int i = -64 -> 0x FF FF FF C0

Little Endian – least significant bit is at lowest address

Memory location	Little-endian	Big-endian
Offset +0	C0	FF
+1	FF	FF
+2	FF	FF
+3	FF	C0

```
$ od -t x4 file1.prdf | more

0000000 a85b0c00 c0ffffff 01000000 001e0300
0000020 08000000 09000000 01000000 001e0300

…

$ od -t x4 file2.prdf | more

0000000 001e0518 ffffffc0 00000001 00008748
0000020 00000008 00000009 00000001 00008748
```

Files with different endianess with a "-64 1" sequence

Variables from files with the wrong endianess need to be byte-swapped

That can be time-consuming!

Have the DAQ write in its native endianess and let the analysis software do the byte-swapping as needed. Don't waste time with that in the DAQ!

Modularity and Extensibility

No one can foresee and predict requirements of a data format 20 years into the future.

Must be able to grow, and be extensible

The way I like to look at this:

FedEx (and UPS) cannot possibly know how to ship every possible item under the sun

But they know how to ship a limited set of box formats and types, and assorted weight parameters

Whatever fits into those boxes can be shipped

During transport, they only look at the label on the box, not at what's inside

We will see a surprisingly large number of similarities with that approach in a minute

“Binary payload agnostic” – what is that?

Most of the “devices” we read out provide their data in some pre-made (and usually quite good) compact binary format already. Usually done in some FPGA.

Actual formatting/packing/zero-suppression in the CPU is rare these days

All you want to do is to grab the blob of data, stick it into a packet, put a label (packet header) on that says what’s in it, done.

That is literally all we do to the data

From that point forward, the DAQ does not care. The “FedEx” approach – they ship boxes, we ship ***packets***.

More generally: Usually we store data from our readout devices, but we must be able to store literally *anything* in our data stream.

Want to store an Excel spreadsheet? A text file? A jpeg image? Shouldn’t cause a problem.

If you think ”why would one want to do that!”, just wait a few minutes.

Example: CAEN's V1742 format

We just take that blob of memory, "put it in a box", done.

The analysis software takes care of the unpacking and interpretation later

Just grab it. Don't waste time here.

3.6. Event structure

An event is structured as follows:

- Header (four 32-bit words)
- Data (variable size and format)

The event can be readout either via VME or Optical Link; data format is 32 bit word.

<table>
<tr><th></th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="4">HEADER</td><td>1</td><td>0</td><td>1</td><td>0</td><td colspan="28">TOTAL EVENT SIZE (LWORDS)</td></tr>
<tr><td colspan="5">BOARD ID</td><td colspan="5"></td><td colspan="14">PATTERN</td><td colspan="4"></td><td colspan="4">GR.MASK</td></tr>
<tr><td colspan="10"></td><td colspan="22">EVENT COUNTER</td></tr>
<tr><td colspan="32">EVENT TIME TAG</td></tr>
<tr><td rowspan="3">GROUP 0</td><td colspan="32">GROUP 0 EVENT DESCRIPTION WORD</td></tr>
<tr><td colspan="32">GROUP 0 CHANNEL DATA</td></tr>
<tr><td colspan="32">GROUP 0 TRIGGER TIME TAG</td></tr>
<tr><td rowspan="3">GROUP 1</td><td colspan="32">GROUP 1 EVENT DESCRIPTION WORD</td></tr>
<tr><td colspan="32">GROUP 1 CHANNEL DATA</td></tr>
<tr><td colspan="32">GROUP 1 TRIGGER TIME TAG</td></tr>
<tr><td rowspan="3">GROUP 2</td><td colspan="32">GROUP 2 EVENT DESCRIPTION WORD</td></tr>
<tr><td colspan="32">GROUP 2 CHANNEL DATA</td></tr>
<tr><td colspan="32">GROUP 2 TRIGGER TIME TAG</td></tr>
<tr><td rowspan="3">GROUP 3</td><td colspan="32">GROUP 3 EVENT DESCRIPTION WORD</td></tr>
<tr><td colspan="32">GROUP 3 CHANNEL DATA</td></tr>
<tr><td colspan="32">GROUP 3 TRIGGER TIME TAG</td></tr>
</table>

How do *we* accomplish that?

The “box” / packet has what I call “envelope information” – a header describing what’s inside

The ***hitformat*** is an enumerated value that determines how the data needs to be unpacked

In PHENIX I have about 200 such formats defined

The packet id uniquely identifies what piece of a given detector this packet holds, or the data from which device

Word	16 bit	16 bit
0	Length	
1	Packet id	Swap unit
2	Hitformat	Padding size
3	Reserved	reserved
4 +	DATA	
n+4	padding	

The order of the packets within an event is irrelevant – a “mini database” – allows to change the read order without breaking anything

Padding – we pad the packet as needed to remain 128-bit aligned

A packet header example (the DRS4 board you see here)

```
$ $ od -j 163888 -t d2 -N 16 beam_0000000042-0000.evt
  0500060    5126        0    1001        4      81       1       0       0

$ dlist beam_0000000042-0000.evt
  Packet  1001  5126 -1 (ONCS Packet) 81 (IDDRS4V1)
```

Word	16 bit	16 bit
0	Length **5126**	
1	Packet id **1001**	Swap unit **4**
2	Hitformat **81**	Padding size **1**
3	Reserved **0**	Reserved **0**
4 +	DATA	
n+4	padding	

This packet says:

I contain data in the hitformat 81 (DRS4 version1 format)

That determines how the data are to be interpreted and decoded later

Data Encapsulation in PHENIX

The unpacker/decoder selected through the hitformat shields the user code from the changing internals of the encoding

The only constant is that the same channels – usually a readout board that we call FEM, Front-End Module – feeds its data into a packet with a never-changing packet id

The packet id identifies a FEM, and a piece of detector "real estate"

It is common to refer to a given FEM by its packet id ("we had a problem with 4033 last night")

But: ***how*** the data are encoded changes over time.

We do not want our analysis code to break because of that!

The packet id tells you *what* is stored in the packet

The hitformat says *how* it is encoded

I can change the encoding for a more efficient one at any time; I just tag it with a new hitfomat (and implement the new decoder acting on that format)

No user software will break!

A real PHENIX event…

This is an actual PHENIX event with the full detector

```
$ dlist  /a/eventdata/EVENTDATA_P00-0000459344-0000.PRDFF
Packet  14001    52  0 (Unformatted)   714 (IDGL1)
Packet  14007    10  0 (Unformatted)   716 (IDNTCZDC_LL1)
Packet  14002     9  0 (Unformatted)   701 (IDBBC_LL1)
Packet  14009    14  0 (Unformatted)   717 (IDGL1_EVCLOCK)
Packet  14011    13  0 (Unformatted)   914 (IDGL1PSUM)
Packet   8180    21  0 (Unformatted)  1508 (IDEMC_FPGA3WORDS0SUP)
Packet   8165    42  0 (Unformatted)  1508 (IDEMC_FPGA3WORDS0SUP)
Packet   8166    48  0 (Unformatted)  1508 (IDEMC_FPGA3WORDS0SUP)
. . .
Packet  25121    83  0 (Unformatted)   425 (IDFVTX_DCM0)
Packet  25122   198  0 (Unformatted)   425 (IDFVTX_DCM0)
Packet  25123    99  0 (Unformatted)   425 (IDFVTX_DCM0)
Packet  25124    46  0 (Unformatted)   425 (IDFVTX_DCM0)
Packet  21351   356  0 (Unformatted)  1121 (IDMPCEXTEST_FPGA0SUP)
Packet  21352   319  0 (Unformatted)  1121 (IDMPCEXTEST_FPGA0SUP)
Packet  21353   238  0 (Unformatted)  1121 (IDMPCEXTEST_FPGA0SUP)
Packet  21354   323  0 (Unformatted)  1121 (IDMPCEXTEST_FPGA0SUP)
```

3827 packets in total in this event

I haven't really mentioned the word "DAQ" yet…

I want to introduce you to my portable DAQ system, "rcdaq" ("really cool data acquisition" – I have a way with names)

What's so cool about it?

The "real" PHENIX DAQ occupies a space about the size of a squash court -- rcdaq is highly portable, lightweight, etc etc – good for ~ 50,000 channels or so, not millions

We use it for R&D, detector commissioning, test beams, what have you

It writes data in the PHENIX format, so the data you take can be analyzed like the real thing

It's a godsend for our students, who usually start out with some test beam data, or work on a detector - the same data format makes for a smooth transition to physics data later

Rcdaq is way more flexible than the big real DAQ and runs on far less demanding hardware

It actually runs on a Raspberry Pi (you can read out the DRS4 evaluation board and some other USB devices)

Example Applications

We are in the middle of upgrading to a new experiment “sPHENIX” – virtually all R&D takes place using RCDAQ as the workhorse DAQ (Stony Brook, Yale, UIUC, BNL…)

The DAQ of the BNL / Stony Brook / UPenn Medical Imaging Groups

A lot of R&D efforts in the Electron-Ion Collider orbit use RCDAQ

One of the DAQ systems available to CERN RD51 collaboration members

Workhorse DAQ at several FermiLab Test Beam setups

Even used in ATLAS for the recent ZDC calorimeter calibration

Medial Department at U of Texas, Arlington

RCDAQ

First off, there are many fine DAQ systems around, some made by people at this school

I’m using mine to show how I implemented the aforementioned principles and some other points

Let me start by asserting that something that just “reads out your detector” does not qualify as a data acquisition system yet – it lives and dies by the amenities it has to offer to really help with your needs.

So what did I implement?

The High Points

Each interaction with RCDAQ is a **shell command**. There is no “starting an application and issuing internal commands” (think of your interaction with, say, root)

RCDAQ out of the box doesn’t know about any particular hardware. All knowledge how to read out something, say, the DRS4 eval board, comes by way of a **plugin** that teaches RCDAQ how to do that.

That makes RCDAQ highly portable and also **distributable** – PHENIX FEMs need commercial drivers for the readout; I couldn’t re-distribute CAEN software, etc etc

RCDAQ has **no configuration** files. (huh? In a minute).

Support for different **event types** (one of the more important features)

Built-in support for standard PHENIX **online monitoring**

Built-in support for an **electronic logbook** (Stefan Ritt’s Elog)

Network-transparent control interfaces

Everything is a shell command…

One of the most important features. Any command is no different from “ls –l“ or “cat”

That makes everything inherently scriptable, and you have the full use of the shell’s capabilities for if-then constructs, error handling, loops, automation, cron scheduling, and a myriad of other ways to interact with the system

Nothing beats the shell in flexibility and parsing capabilities

You can type in a full RCDAQ configuration on your terminal interactively, command by command (although you usually want to write a script to do that)

In that sense, there are no configuration files – only configuration *scripts.*

This is quite different from “my DAQ supports scripts”!

I do not want to be trapped within the limited command set of any application!

As shell commands, the DAQ is fully integrated into your existing work environment

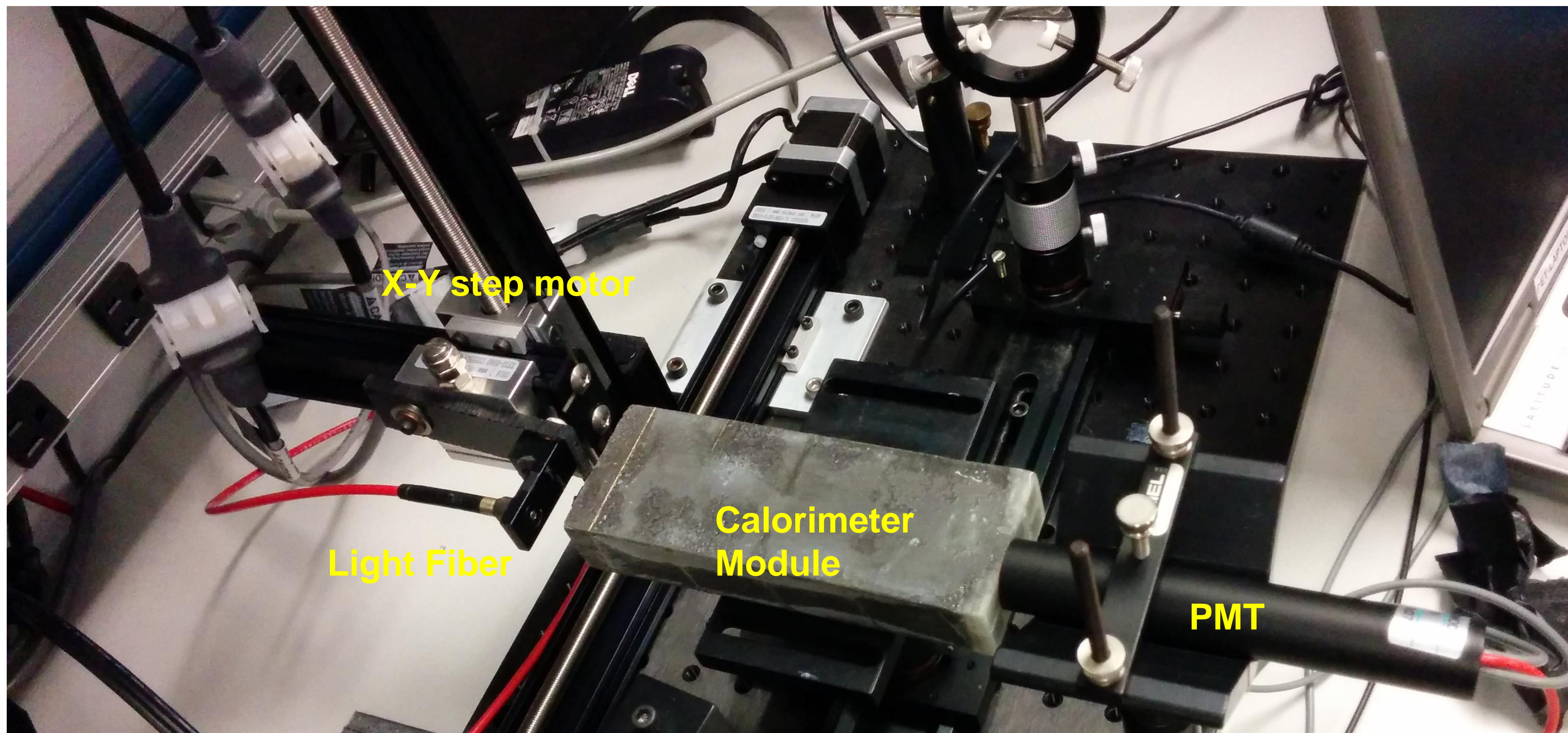
Measurements on autopilot through scripting

You want to run measurements where you step through some values of a parameter completely on autopilot

Here: Move a light fiber with 2 step motors, take a run for each position w/ 4000 events

50 x 25 = 1250 positions (you really want to automate that)

Let it run overnight, come back in the morning, look at the data



The Script

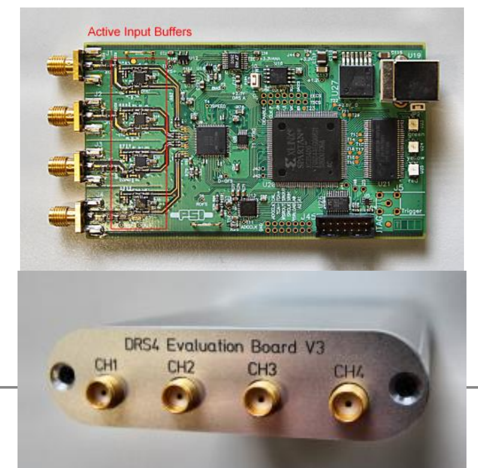
The DAQ operation becomes an integral part of your shell environment

```
#! /bin/sh
STARTPOSX=0
STARTPOSY=9900
INCREMENTX=200
INCREMENTY=-200

CURRENTPOSY=$STARTPOSY
rcdaq_client daq_set_maxevents 4000

for posy in $(seq 25) ; do

    quickmove.sh $CURRENTPOSY 2
    sleep 5
    CURRENTPOSY=$( expr $CURRENTPOSY + $INCREMENTY)
    CURRENTPOSX=$STARTPOSX

    for posx in $(seq 50) ; do

      echo "moving to $CURRENTPOSX"
      quickmove.sh $CURRENTPOSX 1
      sleep 5

      rcdaq_client daq_begin
      wait_for_run_end.sh

      CURRENTPOSX=$( expr $CURRENTPOSX + $INCREMENTX)
    done
done
```

Annotations:

- **Automatic end after 4000 events** → `rcdaq_client daq_set_maxevents 4000`
- 25 positions in y → `for posy in $(seq 25) ; do`
- move the Y motor → `quickmove.sh $CURRENTPOSY 2`
- 50 positions in x → `for posx in $(seq 50) ; do`
- move the x motor → `quickmove.sh $CURRENTPOSX 1`
- **start the DAQ** → `rcdaq_client daq_begin`
- next x → inner `done`
- next y → outer `done`

Setting up and reading out a DRS4 Eval board




```
$ rcdaq_client load librcdaqplugin_drs.so

$ rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3

$ daq_open

$ daq_begin

  # wait a while…

$ daq_end
```

BTW, this double-dash is shell standard for "stop processing command line options"

Else the later "-150" would be interpreted as -1 -5 -0 options like "ls –ltr".

Try to delete a file named "-l"…

You see, each interaction is a separate shell command!

Client-Server Interaction

Think of your session when you use the root package for your analysis

You give commands, use GUIs, and it does what you want

However, you have the exclusive access to your session. No one else (or you in another terminal) can interact with the same root session.

In a DAQ, this is not what one usually wants!

You want more than one “entity” to be able to control your DAQ. Think GUIs, the command line, cron jobs, you name it

Short of control, you want other processes to be able to extract information – extract and display the event rate, the run number, the open file name, etc etc

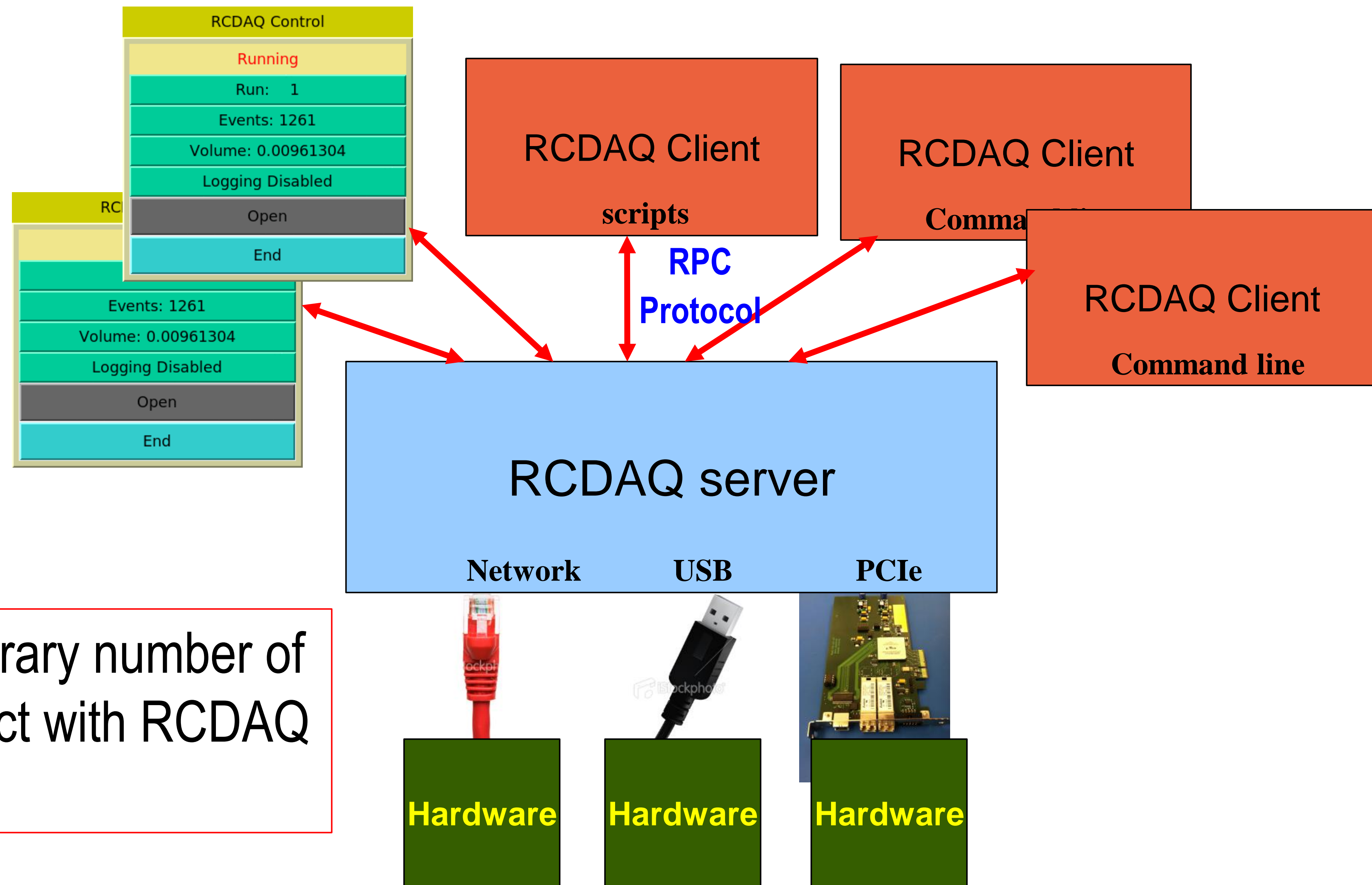
You want a way for more than one process to be able to connect to your DAQ concurrently

The technology I chose is the ***Remote Procedure Call***, RPC

RPC

- Let me first say that there is no shortage of client-server protocols
- CORBA, PVM, there are many others
- The Remote Procedure Call is, in my book, the easiest to use and available everywhere
- Widely established open standard (RFC 1831) for remote execution of code from a client
- Makes it look like a local function call, but the function executes on a server
- Originally meant for off-loading time-consuming functions to a beefy server. We use it to set values and trigger actions in the server.
- The ubiquitous NFS (network file system) is based on RPC, it is available virtually everywhere. Linux. MacOS. Android. Windows. Everything.
- It is a network protocol, so client and server don't have to be on the same machine, can have DAQ and control machine in different rooms (or as far apart as you like as long as the connection traverses the firewalls).

The RCDAQ client-server concept



This allows an arbitrary number of processes to interact with RCDAQ concurrently

The RCDAQ server does not accept *any* input from the terminal. All interaction is through the clients.

```
$ rcdaq_client load librcdaqplugin_drs.so
```

Some standard devices implemented in RCDAQ

RCDAQ

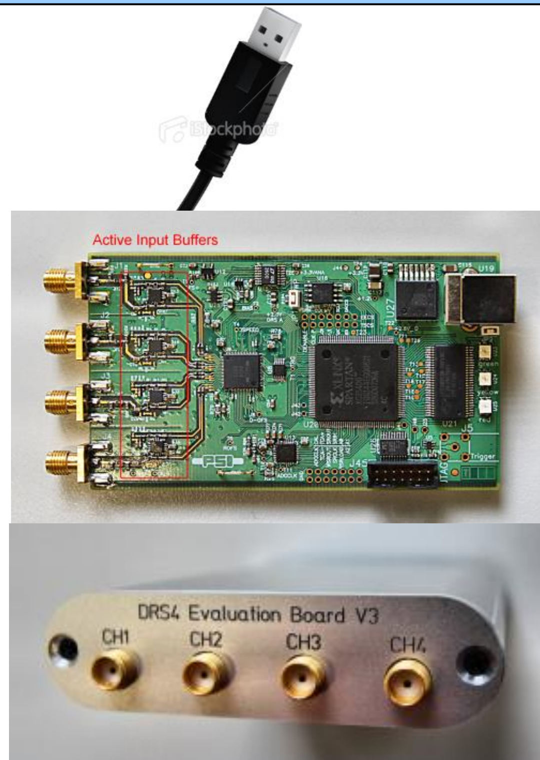
PCIe

PCIe

PCIe

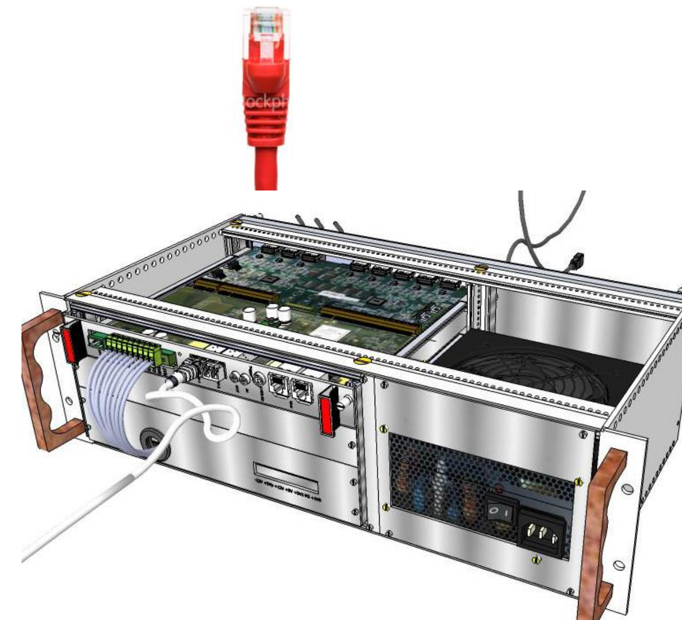
PHENIX Digitizer

new sPHENIX digitizer prototype



DRS4 Eval board

"USB Oscilloscope"



The CERN RD51 SRS System

The CAEN V1742 waveform digitizer

There are many more not shown…

Remember we are talking about a “portable” DAQ here…

Think of a test beam setup (or your Lab setup) for a moment

In the “real” experiment that’s running for a few years (think PHENIX, ATLAS, what have you) you are embedded in an environment that supports all sorts of record keeping

We have the PHENIX run database as an example – we log “everything”, AND there’s infrastructure and suppor it.

I’m not disputing the need f beam needs a different kind of “record keeping support”

What was the temperature? Was the light on? What was the HV? What was the position of that X-Y positioning table?

A database allows you to search for runs with certain properties. But capturing this information in the raw data file is more flexible and **the data cannot get lost**

I often add a webcam picture to the data so we have a visual confirmation that the detector is in the right place, or something

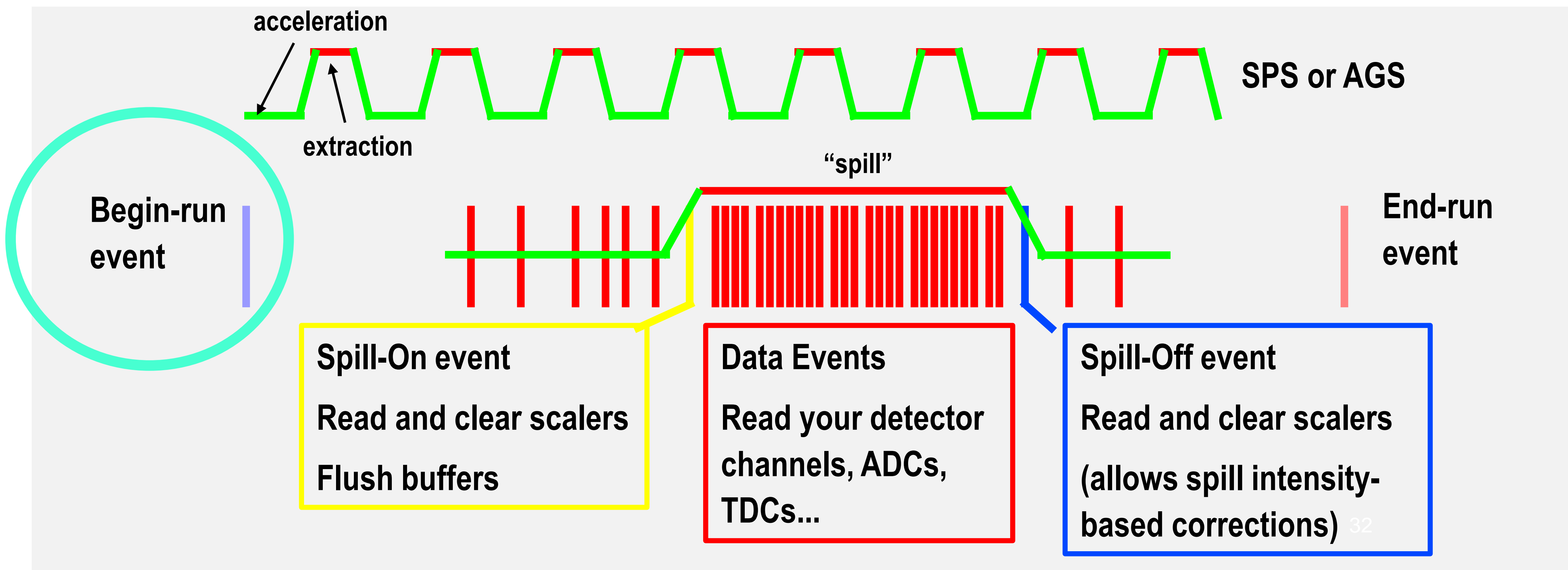
A picture captures everything…

Reading different things with different Event Types

You would think of the DAQ as “reading your detector”

Very often, it is necessary to read different things at different times.

Let’s go to the CERN-SPS (or the BNL AGS) for an example:



In addition to your data, you need information about the spill itself – each one is different

You need to make intensity-dependent corrections on a spill-by-spill basis

So you put some signals on scalers and get an idea about the intensity, dead times, microstructures, etc

Remember this?

This was our typed-in example from before

```
$ rcdaq_client load librcdaqplugin_drs.so

$ rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3
```

A Setup Script

Now you got yourself a setup script as I advertised before, call it, say,

“setup.sh”

```
#! /bin/sh

rcdaq_client load librcdaqplugin_drs.so

rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3
```

Make it executable and you can re-initialize your DAQ each time the same way

Capturing the setup script for posterity

We add this very setup script file into our begin-run event for posterity

This "device" captures a file as text into a packet

This "9" is the event type of the beg-run

And this refers to the name of the file itself

```
#! /bin/sh

rcdaq_client create_device device_file 9 900 "$0"

rcdaq_client load librcdaqplugin_drs.so

rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3
```

So this gets added as packet with id 900 in the begin-run

It's not quite right yet - $0 is usually just "setup.sh", so the server may not be able to find it.

We need the name with a full path!

Expanding the $0 to a full filename

The 3 lines expand the file to a full filename

```
#! /bin/sh

D=`dirname "$0"`
B=`basename "$0"`
MYSELF="`cd \"$D\" 2>/dev/null && pwd || echo \"$D\"`/$B”

rcdaq_client create_device device_file 9 900 "$MYSELF”
rcdaq_client load librcdaqplugin_drs.so
rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3
```

Almost there…

… and the final touch

We clear out any pre-existing device definitions first. We also add some comments as documentation what we are doing here

```
#! /bin/sh
# this sets up the DRS4 readout with 5GS/s, a negative
# slope trigger in channel 1 with a delay of 140

D=`dirname "$0"`
B=`basename "$0"`
MYSELF="`cd \"$D\" 2>/dev/null && pwd || echo \"$D\"`/$B"

rcdaq_client daq_clear_readlist

rcdaq_client create_device device_file 9 900 "$MYSELF"
rcdaq_client load librcdaqplugin_drs.so
rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3
```

More stuff

Most people work from my example scripts that ship with RCDAQ, so it's in in most files…

```
rcdaq_client daq_setrunnumberfile $HOME/.last_rcdaq_runnumber.txt

if ! rcdaq_client daq_status -l | grep -q "CAEN VME1718 Plugin" ; then
    echo "VME1718 plugin not loaded yet, loading..."
    rcdaq_client load librcdaqplugin_caen_vme.so
fi
```

Make run numbers persistent across cold-starts

Figure out if a plugin is loaded and load it if not

You see the beauty of setup scripts with tests, error handling, etc

More special devices

We have seen the device_file, which captures the contents of a file into a packet. What else is there?

device_filenumbers – the “file” saves the contents as text, which is not always easy to work with. Device_filenumbers looks for lines with numbers by themselves on a line, and stores them as numbers. In your analysis, it’s much easier to work with

device_command **-** no packet generated, but an arbitrary command gets executed. (This is one of the most powerful concepts).

device_file_delete – as device_file, but the file gets deleted after inclusion

device_filenumbers_delete – you get the idea

More things from the previous setup

```
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_000001600-0000.evt
8031
8377
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_000001601-0000.evt
8031
8393
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_000001602-0000.evt
8031
8409
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_000001603-0000.evt
8031
8425
```

We are scanning in y direction here

```
rcdaq_client create_device device_file 9 910 /home/eic/struck/positions.txt

rcdaq_client create_device device_filenumbers_delete 9 911 /home/eic/struck/positions.txt

# add the camera picture

rcdaq_client create_device device_command 9 0 "/home/eic/capture_picture.sh
   /home/eic/struck/cam_picture.jpg"

rcdaq_client create_device device_file_delete 9 940 /home/eic/struck/cam_picture.jpg
```

```
8031
8377
```

More about capturing your environment

Many times you capture things only “just in case”

You don’t routinely look at them in your analysis (such a cam pictures shown before)

But if you have some inexplicable feature, you can use the data to do “forensics”

Find out what, if anything went wrong

The more data you capture, the better this gets\

Think of it as “black box” on a plane…

Forensics

"It appears that the distributions change for Cherenkov1 at 1,8,12,and 16 GeV compared to the other energies. It seems that the Cherenkov pressures are changed. […] Any help on understanding this would be appreciated."

Martin: "Look at the info in the data files:"

```
$ ddump -t 9 -p 923 beam_00002298-0000.prdf
S:MTNRG   = -1      GeV
F:MT6SC1 =  5790        Cnt
F:MT6SC2 =  3533        Cnt
F:MT6SC3 =  1780        Cnt
F:MT6SC4 =  0           Cnt
F:MT6SC5 =  73316       Cnt
E:2CH     =  1058  mm
E:2CV     =  133.1 mm
E:2CMT6T =  73.84 F
E:2CMT6H =  32.86 %Hum
F:MT5CP2 =  .4589 Psia
F:MT6CP2 =  .6794 Psia
```

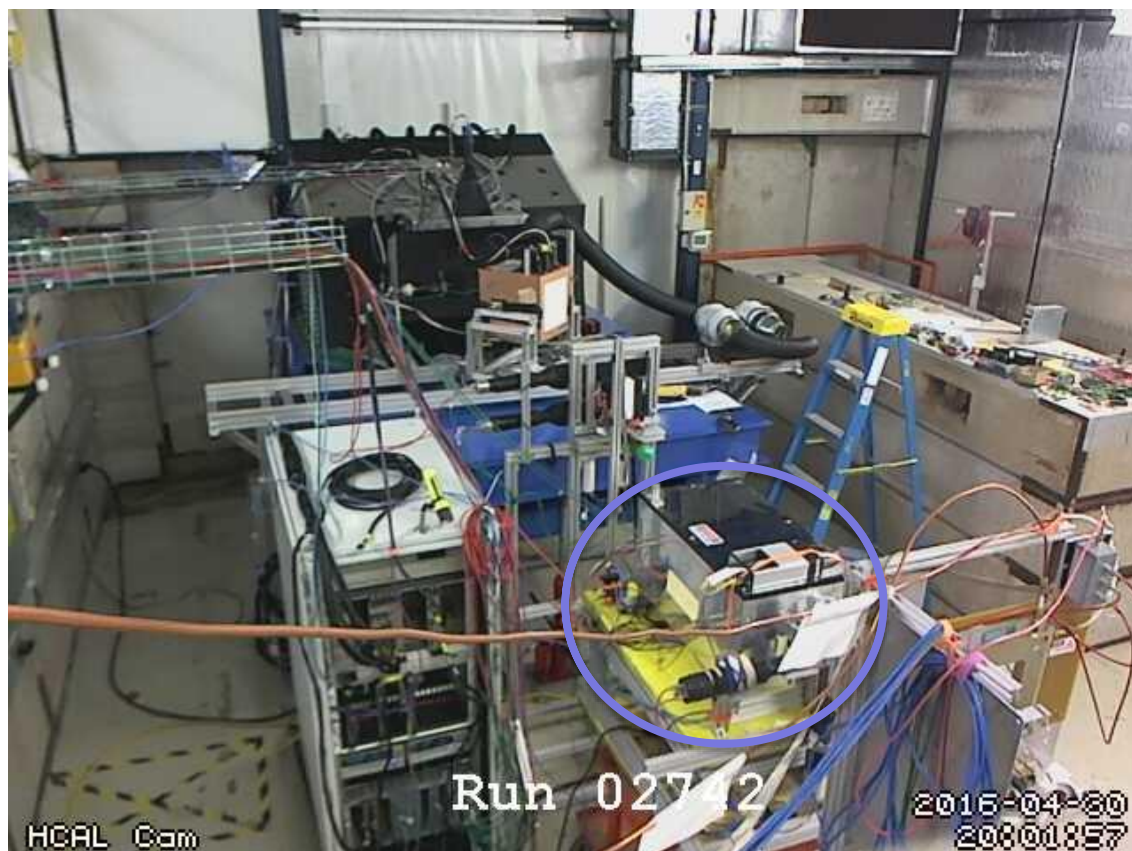
```
$ ddump -t 9 -p 923 beam_00002268-0000.prdf
S:MTNRG   = -2      GeV
F:MT6SC1 =  11846       Cnts
F:MT6SC2 =  7069        Cnts
F:MT6SC3 =  3883        Cnts
F:MT6SC4 =  0           Cnts
F:MT6SC5 =  283048      Cnts
E:2CH     =  1058  mm
E:2CV     =  133   mm
E:2CMT6T =  74.13 F
E:2CMT6H =  37.26 %Hum
F:MT5CP2 =  12.95 Psia
F:MT6CP2 =  14.03 Psia
```

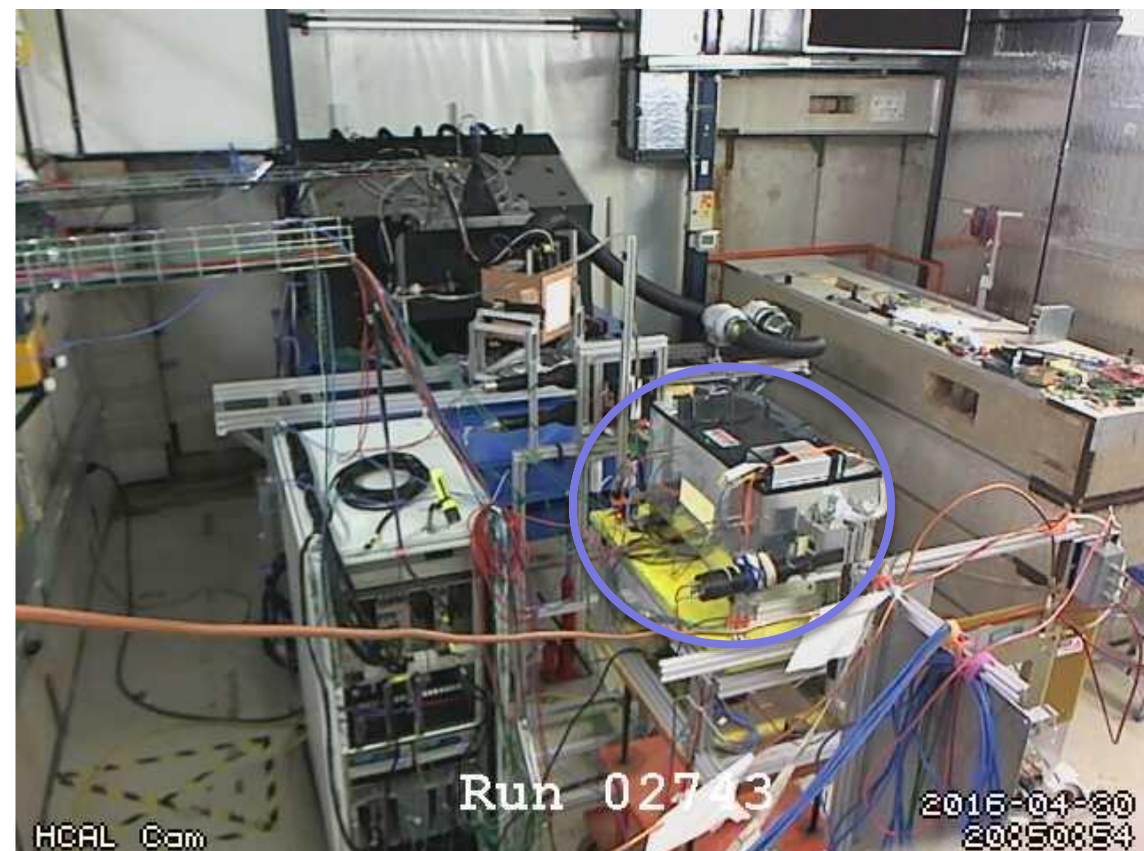
More Forensics

"Did that other group's contraption get moved into the beam in run 2743? There is a higher fraction of showering than before."

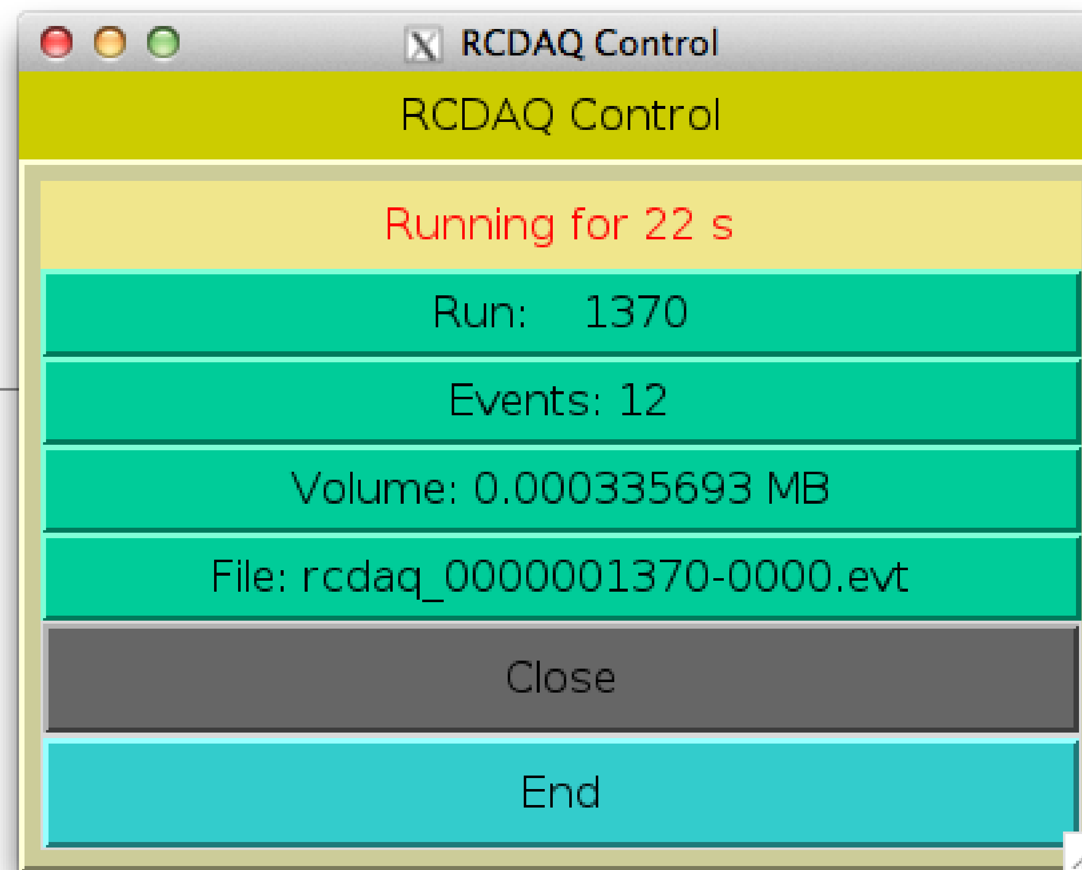
Look at the cam pictures we automatically captured for each run:

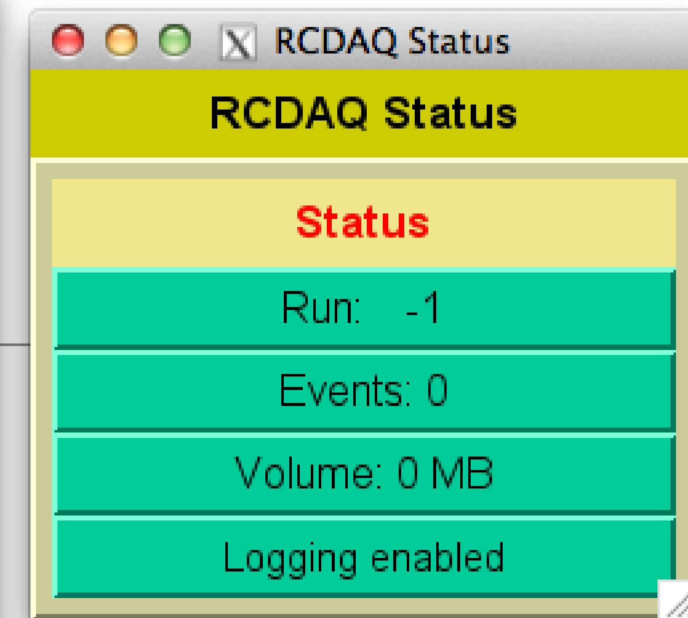
```
$ ddump -t 9 -p 940 beam_00002742-0000.prdf > 2742.jpg
$ ddump -t 9 -p 940 beam_00002743-0000.prdf > 2743.jpg
```





GUIs





- **GUIs must not be stateful!**
- Statelessness allows to have multiple GUIs at the same time
- And allows to mix GUIs with commands (think scripts)
- (all state information is kept in the rcdaq server)
- My “GUI” approach is to have perl-TK issue standard commands, parse the output
- Special no-frills daq_status -s (“short”) geared towards easy parsing

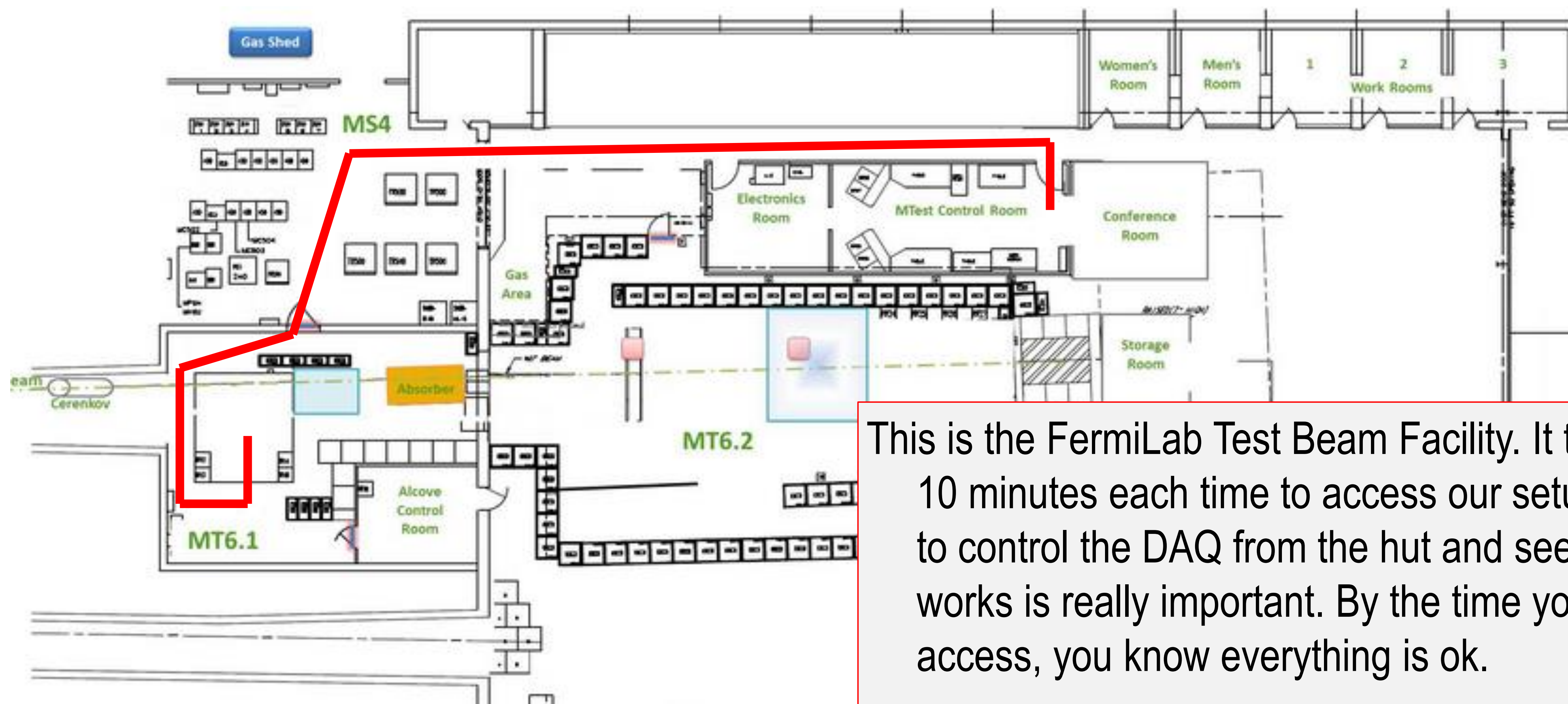
Why stateless GUIs (and controls in general)

They allow you to run any number of GUIs

You can enter RCDAQ commands from any terminal that can reach the DAQ machine

Say you fix something at your setup – you can control the remote DAQ from your laptop that you brought with you for the access

Also remember that the controls travel through the network



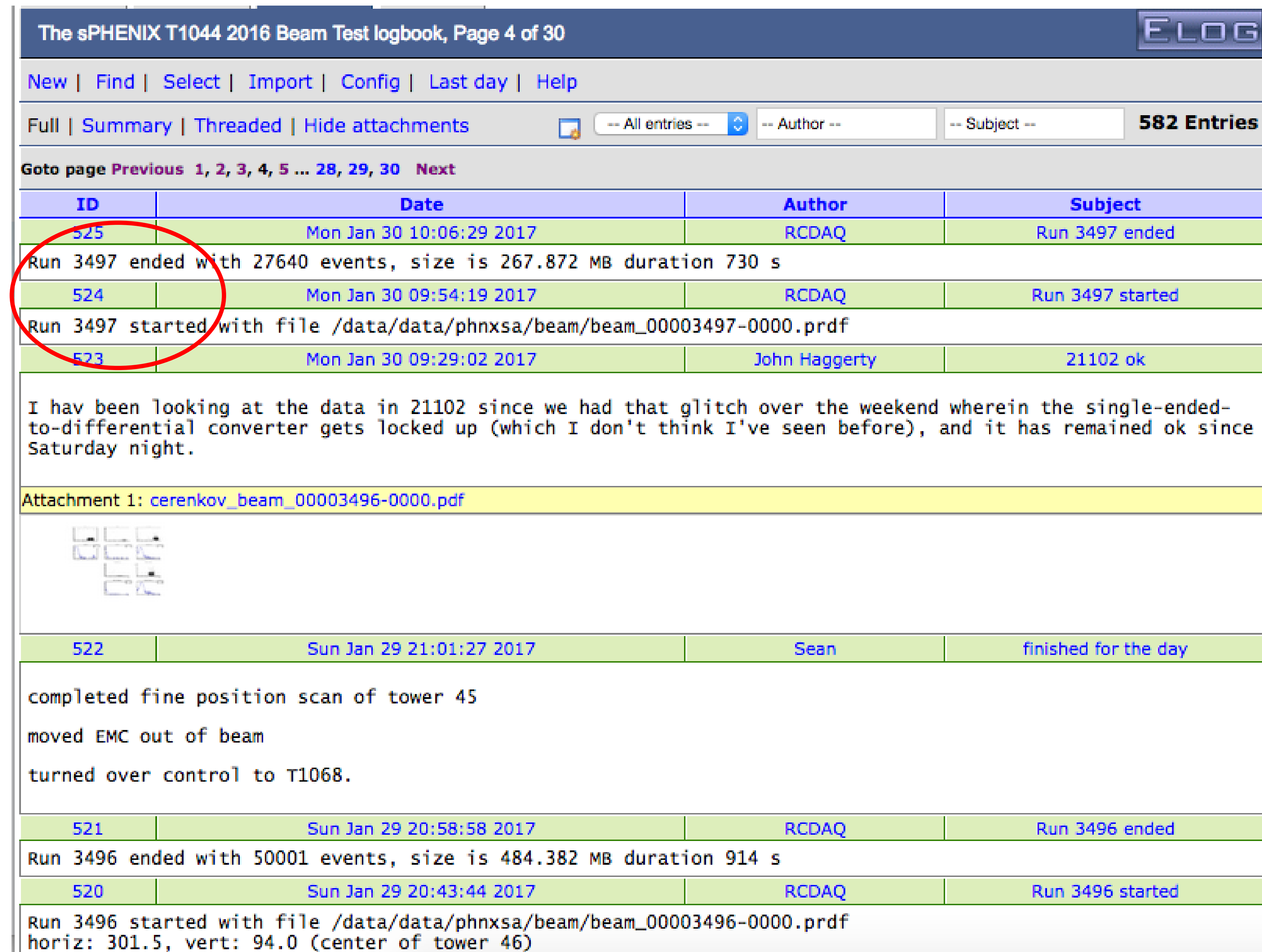
This is the FermiLab Test Beam Facility. It took us about 10 minutes each time to access our setup. The ability to control the DAQ from the hut and see that everything works is really important. By the time you end the access, you know everything is ok.

Automated Elog Entries

RCDAQ can make automated entries in your Elog

Of course you can make your own entries, document stuff, edit entries

Gives a nice timeline and log



Coming back to the “shell command” feature

For the last 3 minutes, I want to harp some more on the superiority of that “everything is a shell command” approach

Often I’m learning of a new ingenious way to use this aspect for something cool

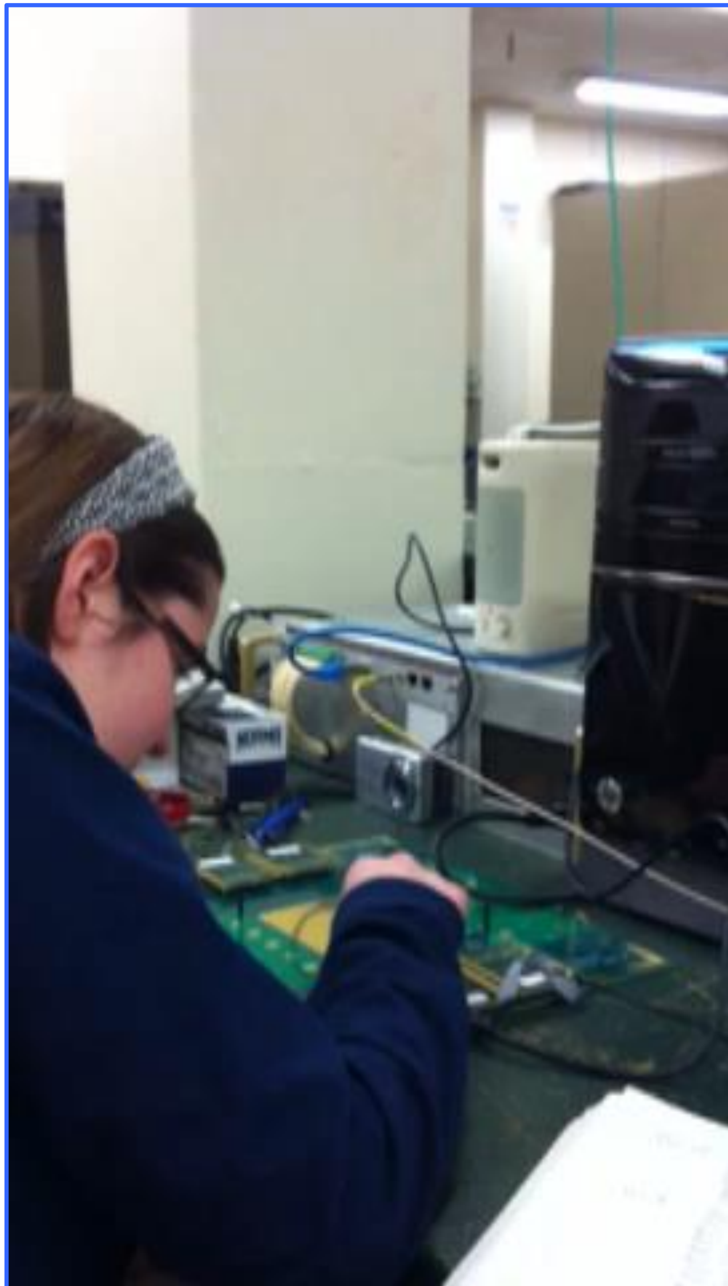
A real good tool gets used in ways that the designer did not envision… but it works!

A group needed to test a few thousand pads on a plane if they a) work and b) are connected right.

Inject charge into the pads one by one... **but you can't take your eyes (or the probe) off the pad plane or you lose your position**

They came up with…

Shell integration



THE SPEAKING DAQ

```
#! /bin/sh

rcdaq_client daq_setfilerule /home/sbeic/calibfiles/srs-%010d-%02d.evt

for column in $(seq $1 $2) ; do

    for row in $(seq 0 20) ; do
        echo "$column and row $row" | festival --tts
        sleep 2

        echo "Go" | festival --tts

        echo rcdaq_client daq_begin  ${column}555${row}
        rcdaq_client daq_begin  ${column}555${row}

        sleep 3
        echo "End" | festival --tts

        echo rcdaq_client daq_end
        rcdaq_client daq_end


    done
done

rcdaq_client daq_setfilerule /home/sbeic/datafiles/srs-%04d-%02d.evt
```

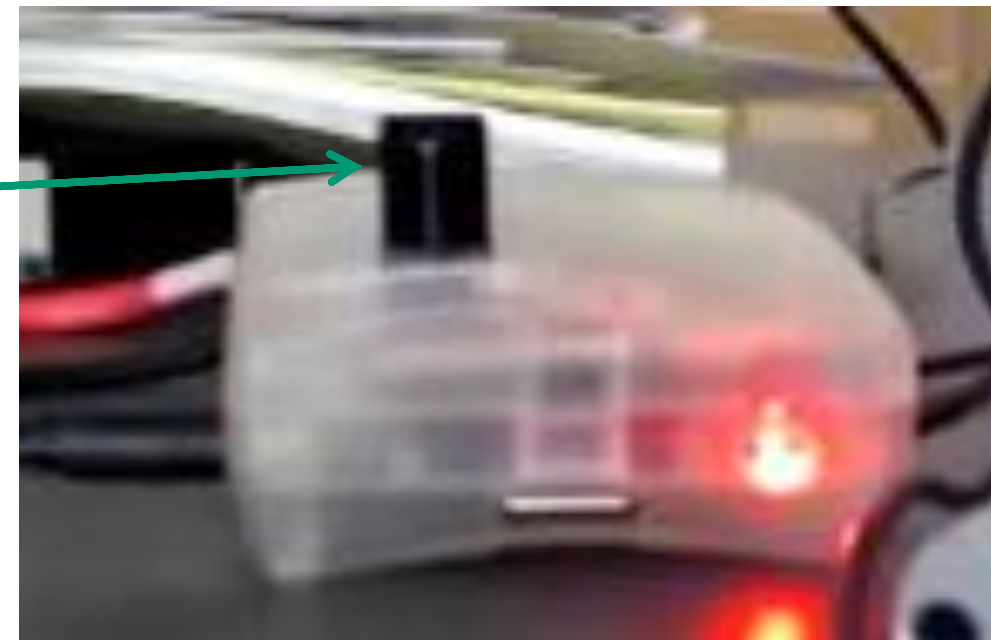
One final cool thing

Anything that's capable of issuing a shell command can control the DAQ

I have said (but not shown you yet) that the DAQ can be controlled remotely, through the network

I have a Raspberry Pi connected here that I have set up so it controls RCDAQ running on my Laptop

And you see it has developed some kind of growth on its head… That's an infrared receiver



We know we can assign arbitrary commands to buttons pressed on virtually any IR remote

I guess you see where this is going…

Summary

I used RCDAQ to show some design principles

I want to re-iterate that there are many fine DAQ systems “out there”

We have seen the virtues of shell-command only interactions

Learned about Event Types for different cool things, especially the begin-run

We learned why we want stateless GUIs and commands, and be network-transparent

We didn’t have time to talk about the online monitoring, but it’s there

Also, there’s still quality time during the school

A portable DAQ system – in a minute you will see me carry it back to my seat.

THANK YOU!

This page is intentionally left blank