RCDAQ - A scalable, portable DAQ system design

Martin L. Purschke

BROOKHAVEN NATIONAL LABORATORY

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

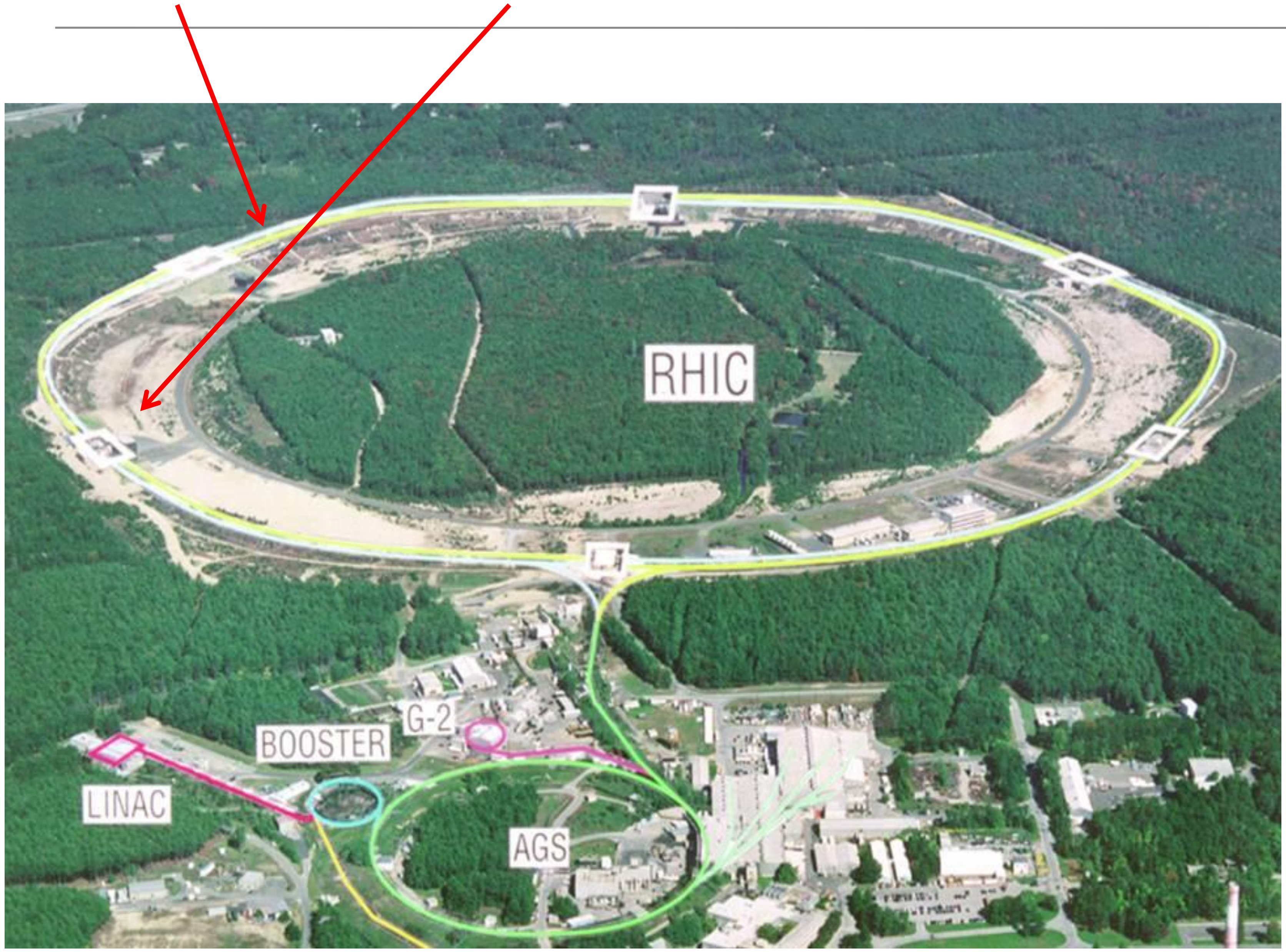
Moved to Brookhaven National Laboratory to work with the Relativistic Heavy Ion Collider

Am the DAQ coordinator for the PHENIX Experiment

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”

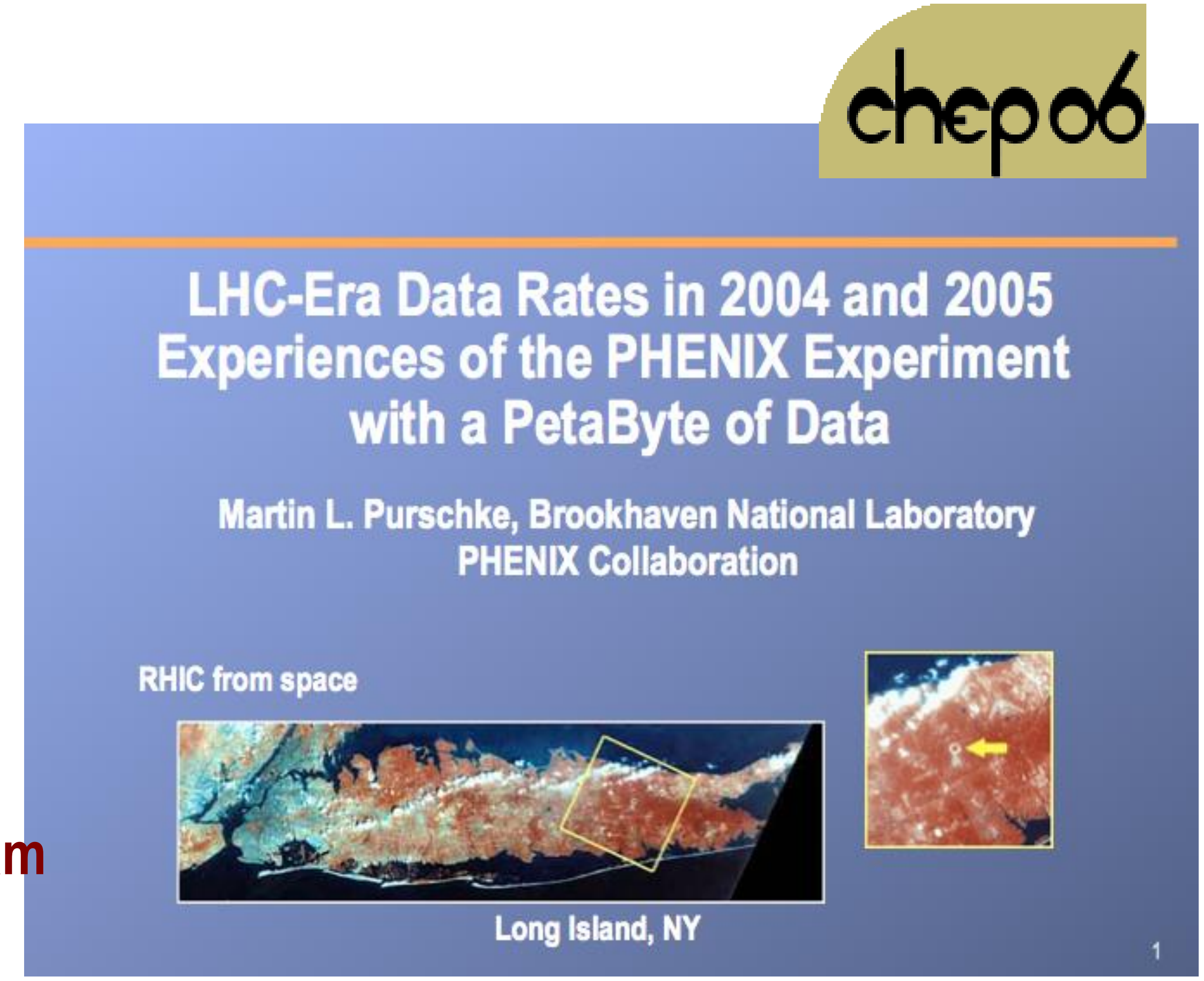
So a trigger can be truly selective

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



chep06

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

We will be using the RCDAC data acquisition system here in several places at the school

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

Will tell you about the basics

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

This is by no means the only or “the best” data acquisition system. People in this room have built DAQ systems which are in widespead use,

Notably, Stefan’s MIDAS system….

RCDAQ is just what we will be using.

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

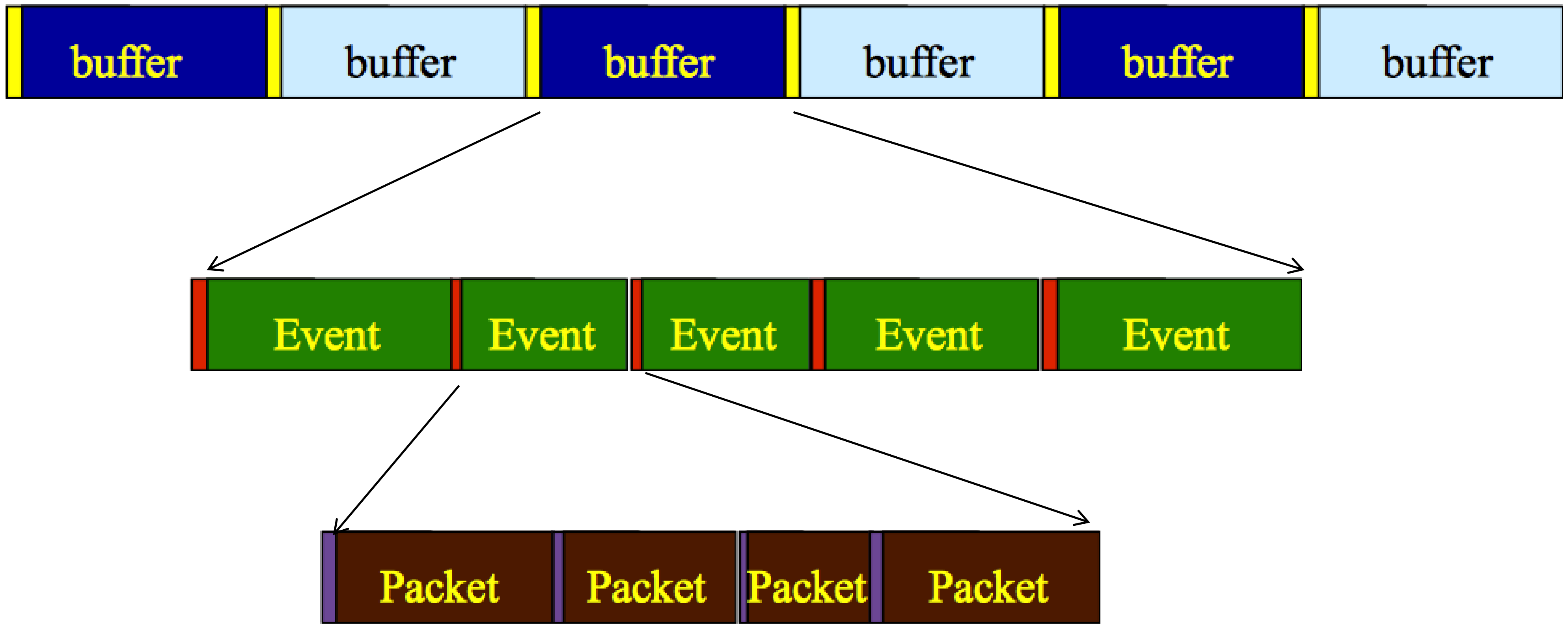
"*You must be able to erroneously feed your mail file to your analysis. It shouldn't find events, but it shouldn't crash, either.*"

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 even, 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 multiples 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 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 $\longrightarrow$

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.

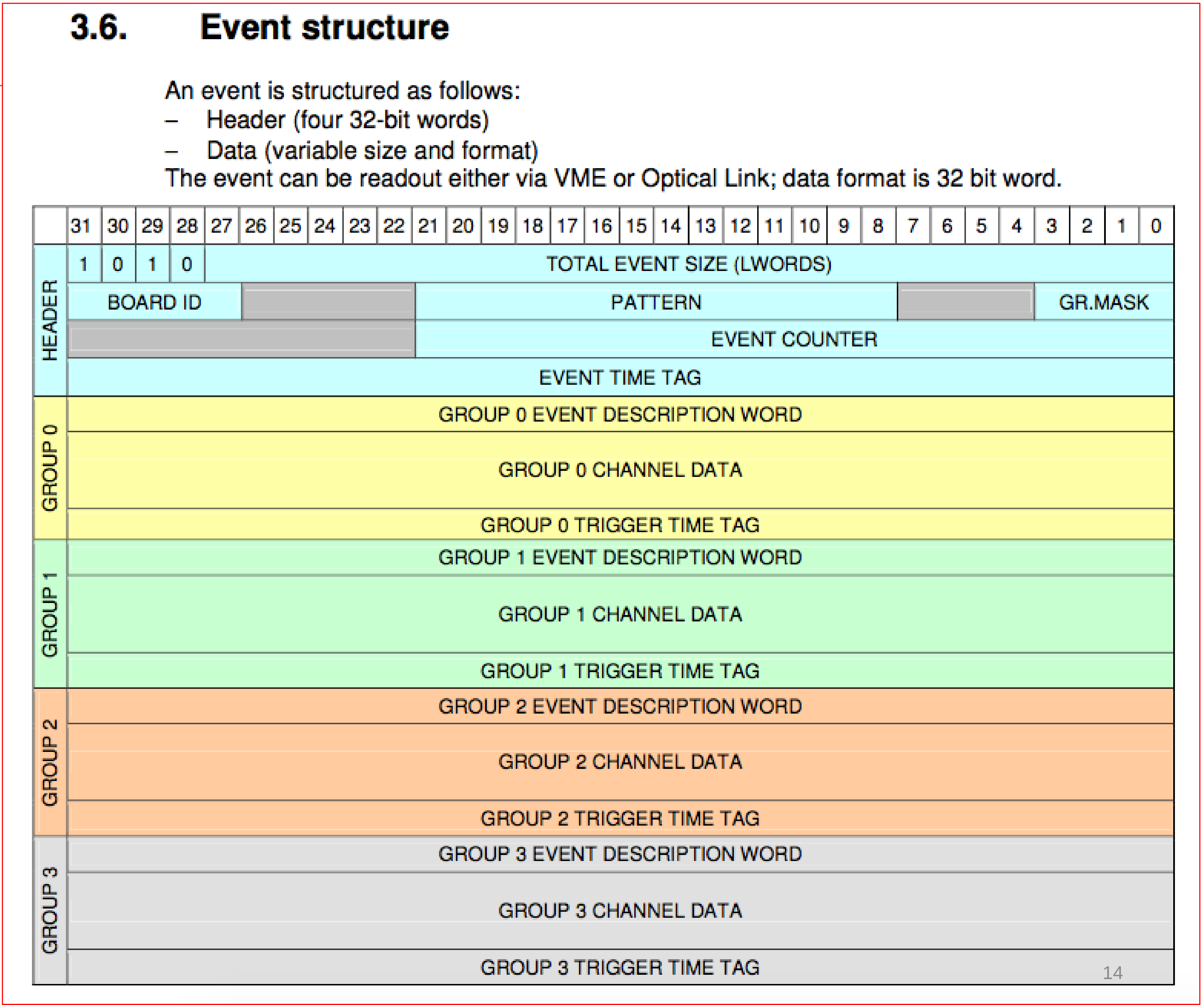
If you think ”why would one want to do that!”, just wait a few minutes.

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.

	31	30	29	28	27–0
HEADER	1	0	1	0	TOTAL EVENT SIZE (LWORDS)
HEADER	BOARD ID				▨ / PATTERN / ▨ / GR.MASK
HEADER	▨				▨ / EVENT COUNTER
HEADER	EVENT TIME TAG				
GROUP 0	GROUP 0 EVENT DESCRIPTION WORD				
GROUP 0	GROUP 0 CHANNEL DATA				
GROUP 0	GROUP 0 TRIGGER TIME TAG				
GROUP 1	GROUP 1 EVENT DESCRIPTION WORD				
GROUP 1	GROUP 1 CHANNEL DATA				
GROUP 1	GROUP 1 TRIGGER TIME TAG				
GROUP 2	GROUP 2 EVENT DESCRIPTION WORD				
GROUP 2	GROUP 2 CHANNEL DATA				
GROUP 2	GROUP 2 TRIGGER TIME TAG				
GROUP 3	GROUP 3 EVENT DESCRIPTION WORD				
GROUP 3	GROUP 3 CHANNEL DATA				
GROUP 3	GROUP 3 TRIGGER TIME TAG				

Bit columns: 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

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

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!

Data Encapsulation – changing encoding

Example: our Muon Tracker delivers 5 10bit values per channel.

Until 2006 or so, we would stick each 10bit value into a 16bit word, so 100 channels => 500 values => 1000 bytes

Then we would use 4 bytes to store 3 values, so 100 channels occupy 750 bytes now

Does the analysis code need to change? **No**.

A new hitformat selects a different unpacker / decoder for that new format which delivers the decoded data just as before

All invisible to the user code - no code can break because of an encoding change.

The threshold to change the encoding isn't super-high because of that

We commissioned a new detector in 2014 and are on its 3rd hitformat because we found that we'd need addt'l information to better understand the detector. Whatever existed still works as before.

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 Stefan’s DRS4 board and some other USB devices)

RCDAQ

I'm using my creation to show how I implemented the aforementioned principles and some other points

It can read thousands of channels on a fast machine, but is lighweight enough that it runs on a Raspberry Pi

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

I decided to make each interaction with RCDAQ 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 does not have configuration files. (huh? In a minute).

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

Built-in support for online monitoring

Built-in support for an electronic logbook (Stefan’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.*

Before we get to the technology, let’s set up a DRS4 for readout, interactively

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 885

$ daq_open

$ daq_begin

  # wait a while…

$ daq_end
```

You see, each interaction is a separate shell command

“daq_open” is actually an alias to “rcdaq_client daq_open”, etc

When there is a client, there is a server…

And that brings us to the choice of technology I used in RCDAQ.

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 technologyI 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).

How does this work?

```
int main()
{
  light_zero(1);  // turn on light number 0
  sleep(5);       // sleep 5s
  light_zero(0);  // and turn it off again
  return 0;
}
```

This is a simple main program which turns on a light for 5s.

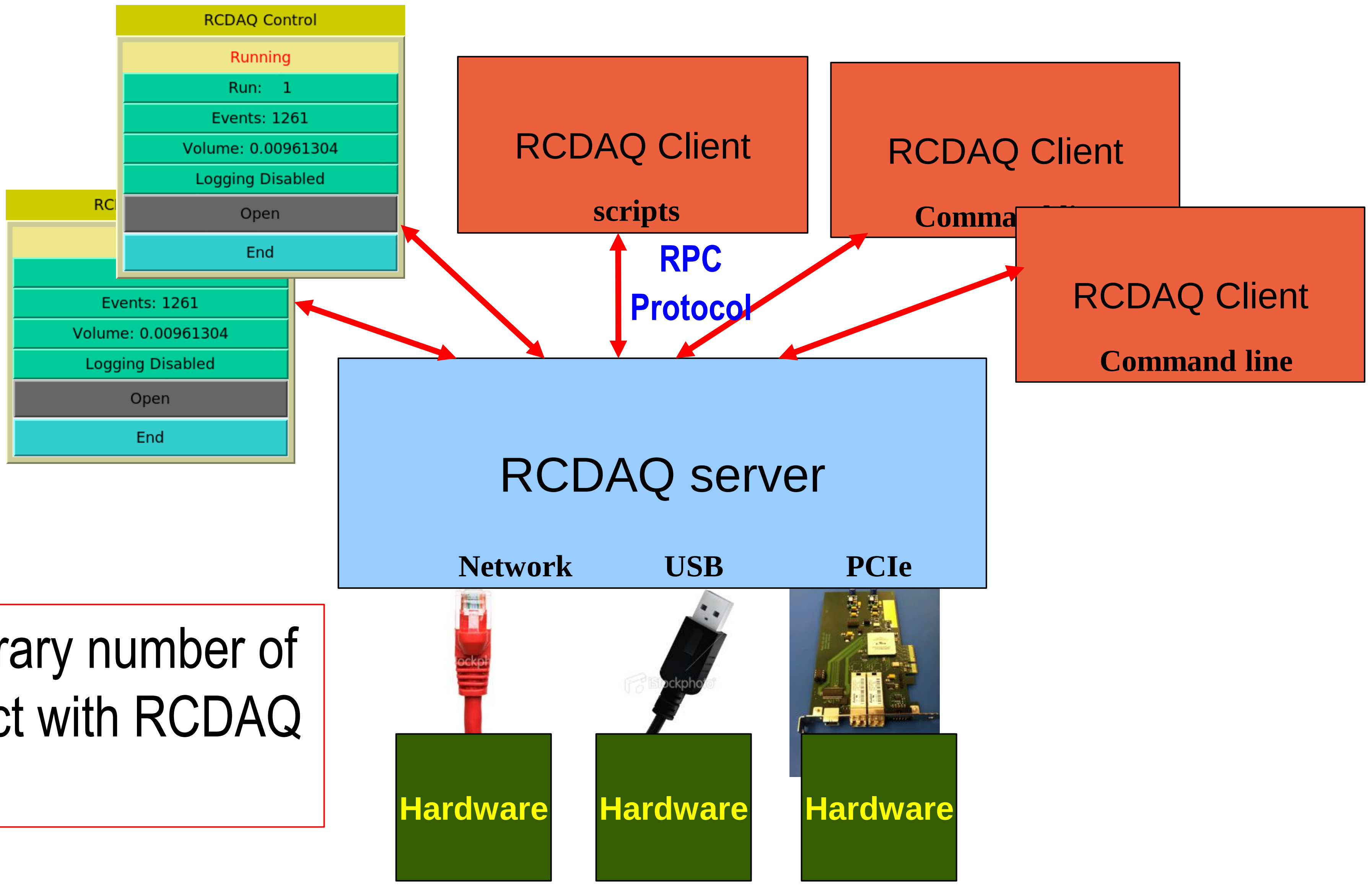
```
program REMOTE
{
  version REMOTE_VERS
    {
      int light_zero( int i) = 1;
      int light_one( int i) = 2;
    } = 1;
} = 0x3feeef29;
```

This RPC definition can generate code that you make the calls available as a server, and you can build clients that then call those remote functions

This is an example where you set values (lights on or off) in a server

The RPC machinery makes that painless (code generation, network interaction, etc)

The RCDAQ client-server concept

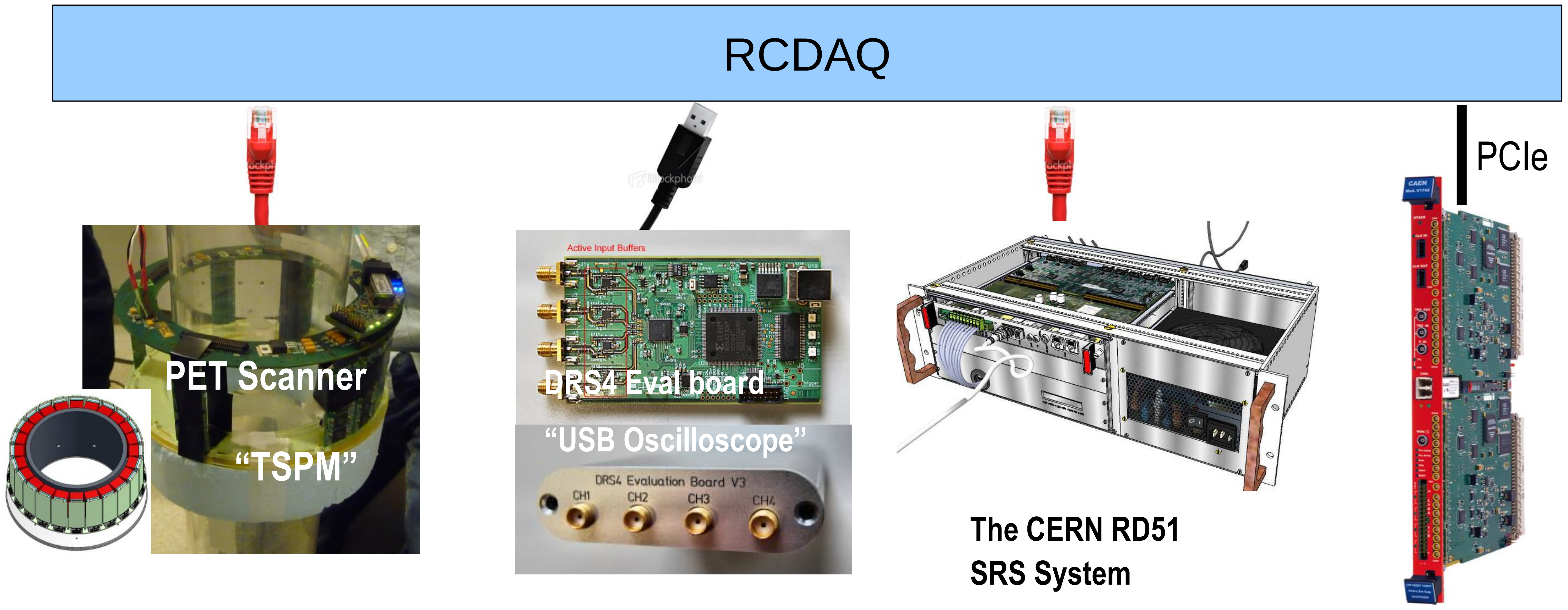


This allows an arbitrary number of processes to interact with RCDAQ concurrently

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



The CERN RD51 SRS System

The CAEN V1742 waveform digitizer

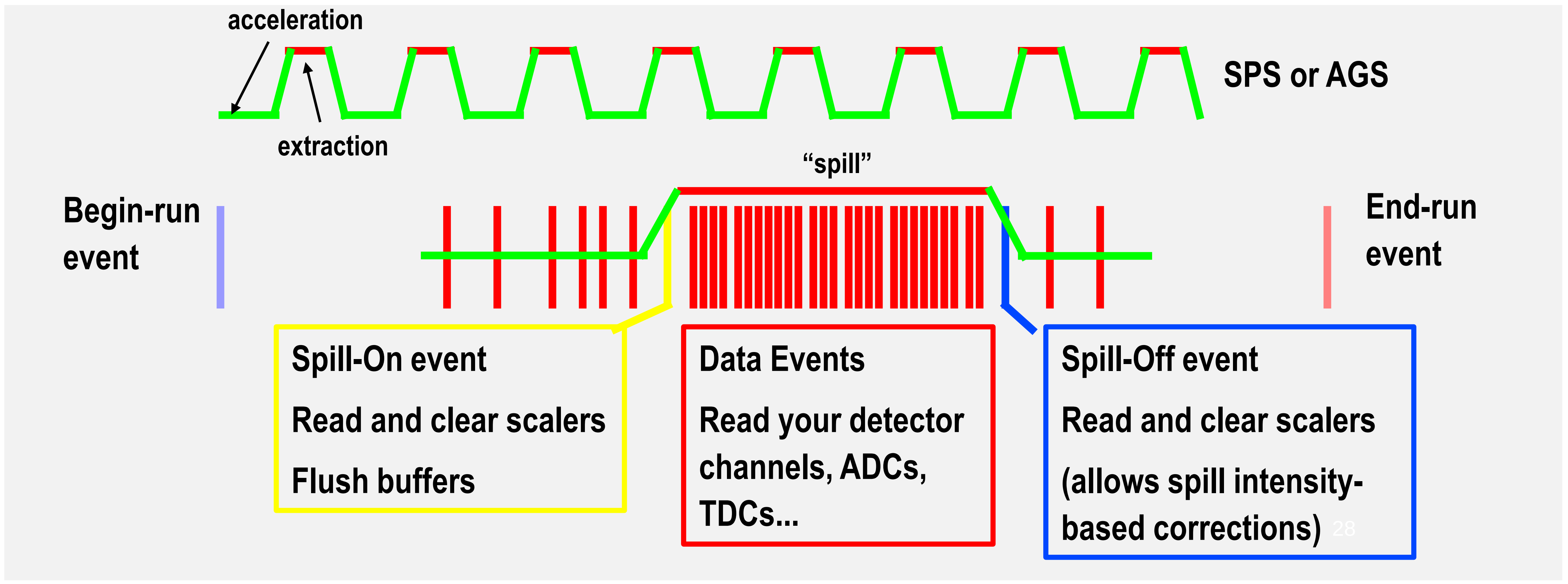
There are more not shown…

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:



Spill-On event

Read and clear scalers

Flush buffers

Data Events

Read your detector channels, ADCs, TDCs...

Spill-Off event

Read and clear scalers

(allows spill intensity-based corrections)

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

Those Different Event Types and what they are good for

“reading out your detector” does not yet make a data acquisition

It lives and dies by its ability to capture... well, everything

What was the HV? Was the light on? What was the temperature?

I'll try to show you that anything - and I mean anything - that is known to the DAQ or any other computer can be captured in the raw data stream

In many cases, it's there only for forensics in case there's something wrong

Was the HV where it was supposed to be?

Often, you can get parameters which in the old days you had to type in from a paper logbook

Test beams often involve moving the detector in the beam, so capture the position

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

Especially the Begin-Run event is a good place to store as many relevant parameters as you can

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 885
```

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 885
```

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 885
```

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.

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 885
```

Almost there…

… and the final touch

We clear out any pre-existing device definitions first:

```
#! /bin/sh

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 885
```

Now it is safe to execute this script more than once, for example, if you make changes.

At any time, you will be able to look at the way RCDAQ was set up from the begin-run event.

Self-documenting your DAQ

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

```
$ ddump -t 9 -p 900 -O Xray_HVscan_0000001900-0000.evt

#! /bin/sh

if ! rcdaq_client daq_status > /dev/null 2>&1 ; then

    echo "No rcdaq_server running, starting..."

    rcdaq_server > $HOME/log/rcdaq.log 2>&1 &

    sleep 2

    ELOG=$(which elog 2>/dev/null)

    [ -n "$ELOG" ]  && rcdaq_client elog localhost 666 RCDAQLog

fi

# many more lines deleted…
```

This command shows the contents of packet 900 from the begin-run

This if block checks if the server is already running and starts it if not

Also sets up the Elog interaction

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
```

Make run numbers persistent across cold-starts

```
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
```

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_0000001600-0000.evt
8031
8377
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_0000001601-0000.evt
8031
8393
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_0000001602-0000.evt
8031
8409
eicdaq2 ~ $ ddump -O -p 910 -t 9  ZZ48_0000001603-0000.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 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
```

File Rules

The output files are generated according to a file rule that you can set

This is just a plain “printf” control string that takes two numbers

Default is rcdaq-%08d-%04d.evt

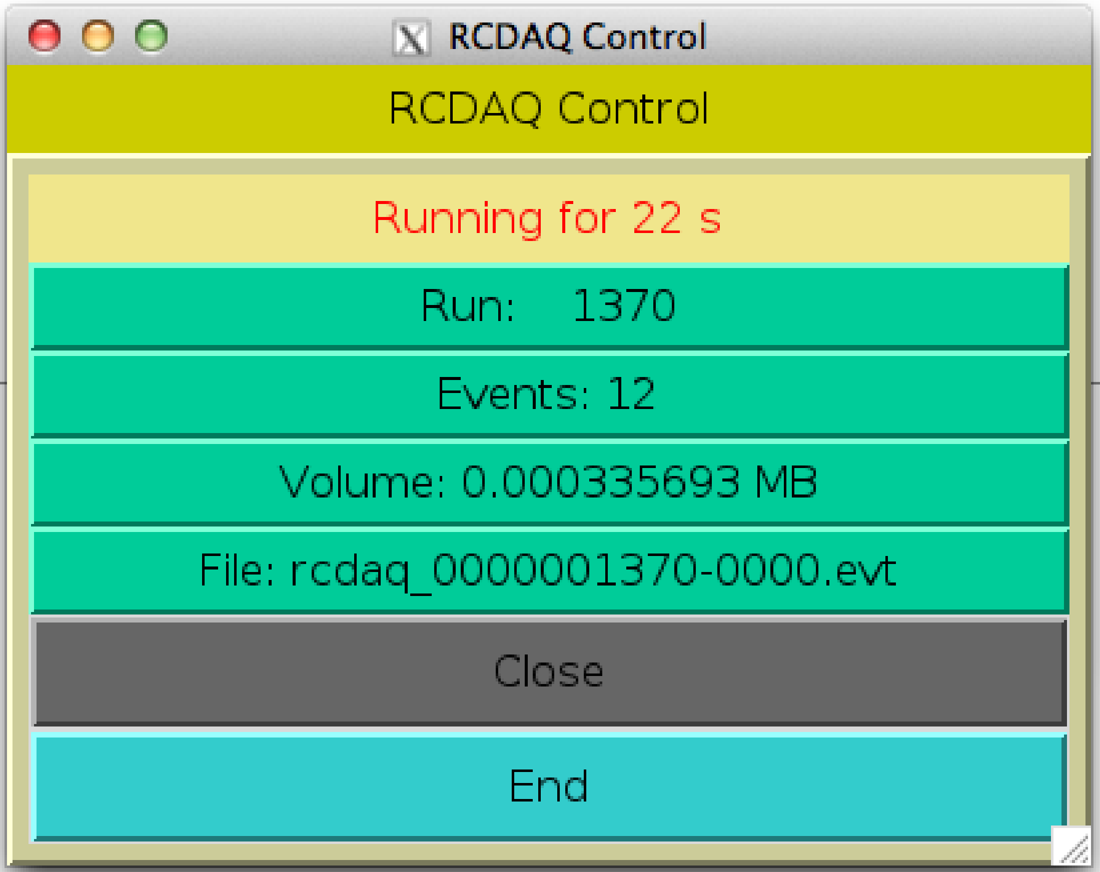
Takes run number and “file sequence number” - the latter is for rolling over the file at a predetermined size so any one file doesn't get too large

For example for “run 1234” :

```
$ printf "rcdaq-%08d-%04d.evt\n" 1234 0
rcdaq-00001234-0000.evt
```

You can change the rule at any time.

GUIs


RCDAQ Control

RCDAQ Control

Running for 22 s

Run: 1370

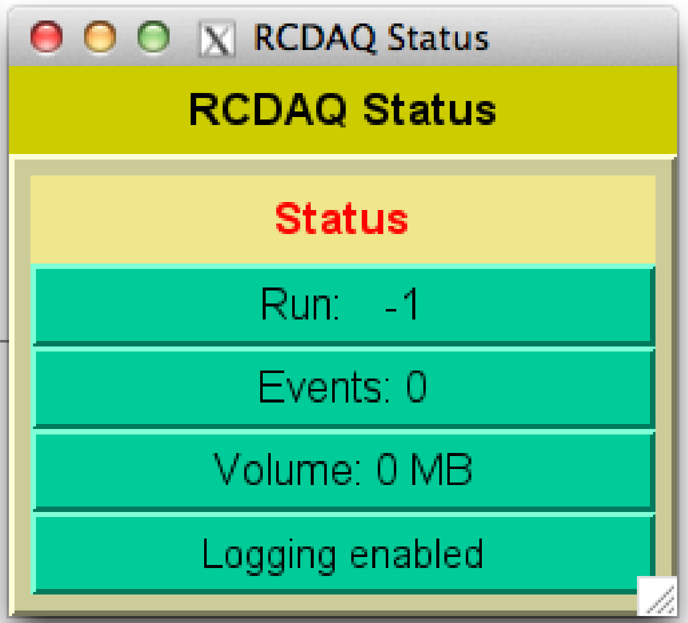
Events: 12

Volume: 0.000335693 MB

File: rcdaq_0000001370-0000.evt

Close

End


RCDAQ Status

RCDAQ Status

Status

Run: -1

Events: 0

Volume: 0 MB

Logging enabled

- **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

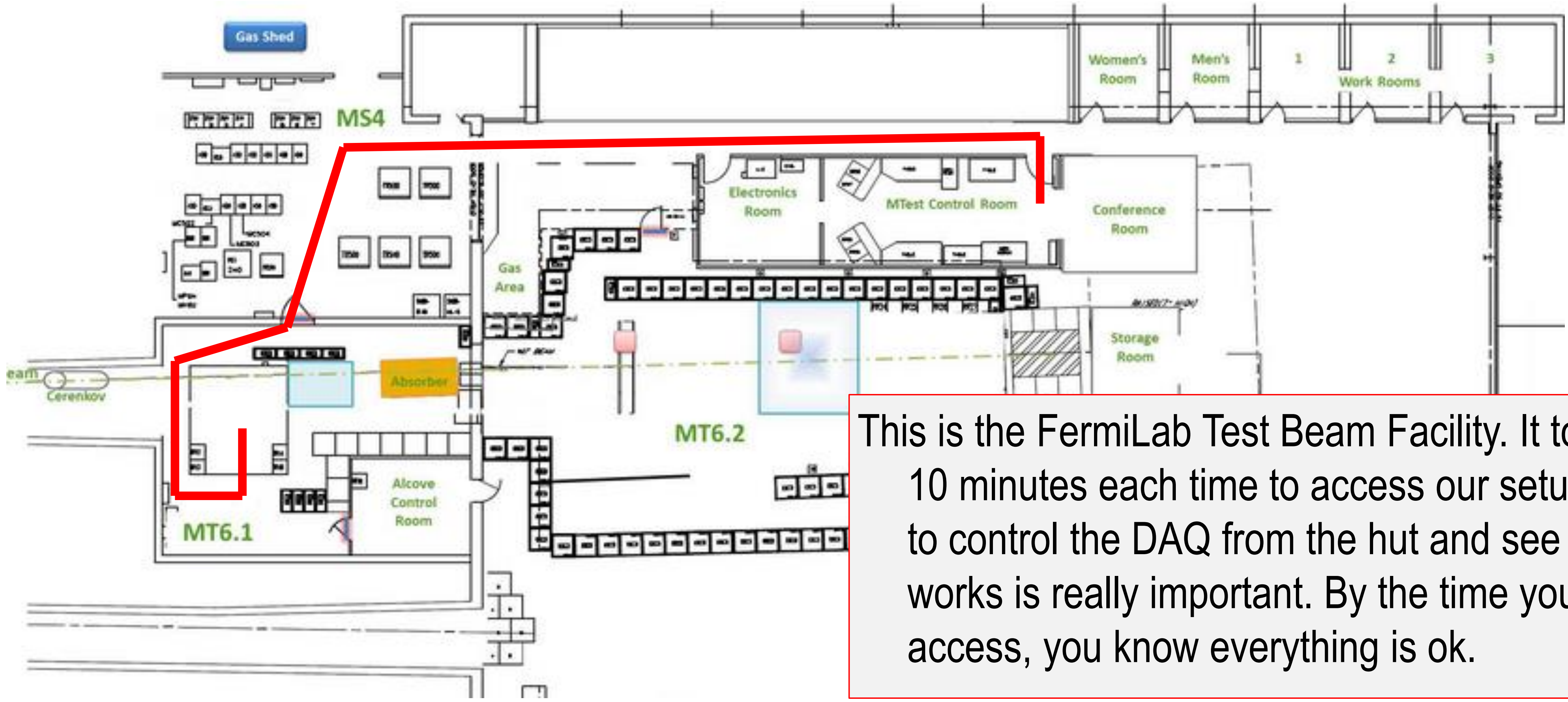
Why stateless GUIs (and controls in general)

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



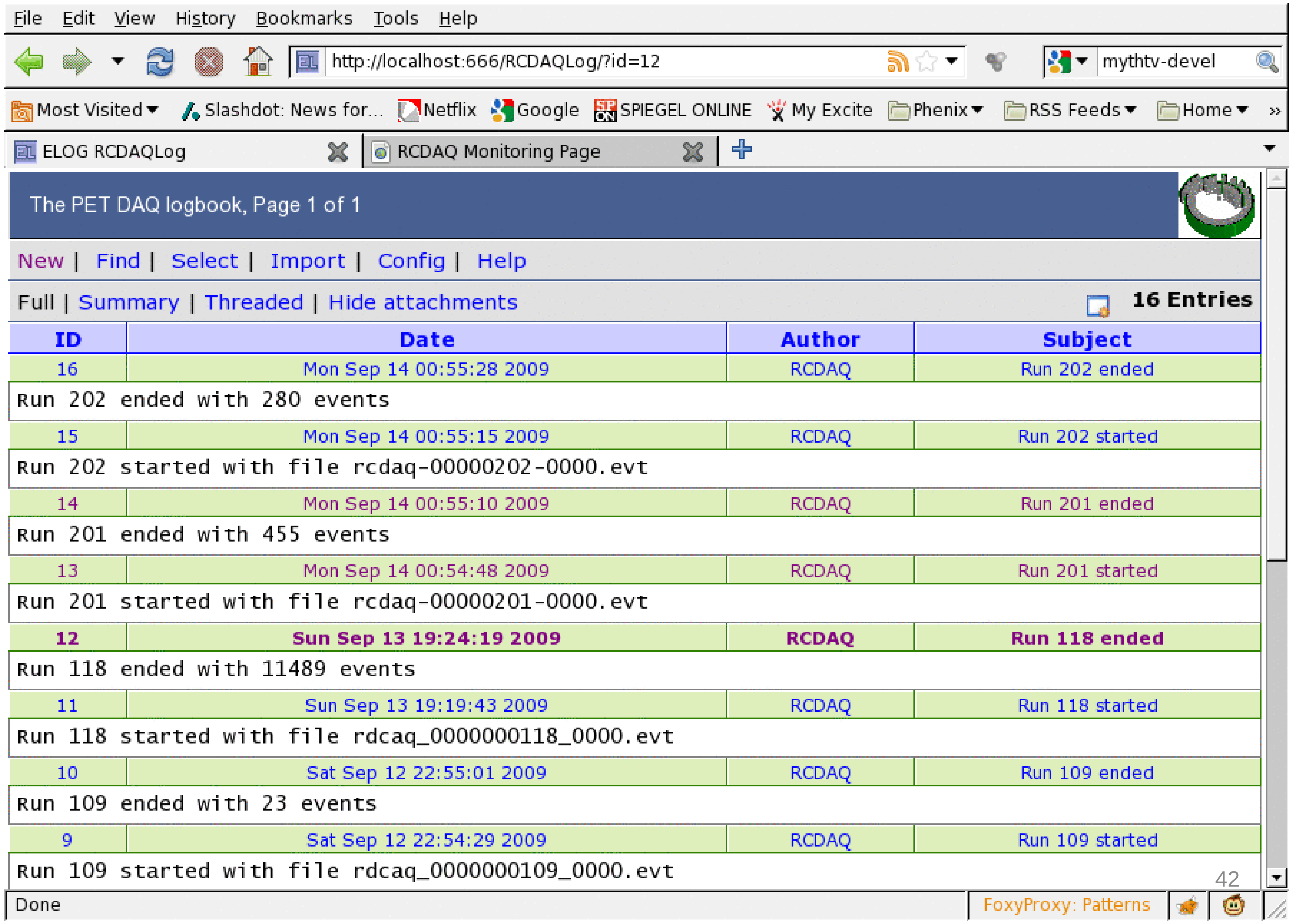
This is the FermiLab Test Beam Facility. It took us about 10 minutes each time to access our setup. The ability to control the DAQ from the hut and see that everything works is really important. By the time you end the access, you know everything is ok.

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


File Edit View History Bookmarks Tools Help
http://localhost:666/RCDAQLog/?id=12
mythtv-devel
Most Visited Slashdot: News for... Netflix Google SPIEGEL ONLINE My Excite Phenix RSS Feeds Home
ELOG RCDAQLog
RCDAQ Monitoring Page
The PET DAQ logbook, Page 1 of 1
New | Find | Select | Import | Config | Help
Full | Summary | Threaded | Hide attachments
16 Entries
Done
FoxyProxy: Patterns

ID	Date	Author	Subject
16	Mon Sep 14 00:55:28 2009	RCDAQ	Run 202 ended
Run 202 ended with 280 events			
15	Mon Sep 14 00:55:15 2009	RCDAQ	Run 202 started
Run 202 started with file rcdaq-00000202-0000.evt			
14	Mon Sep 14 00:55:10 2009	RCDAQ	Run 201 ended
Run 201 ended with 455 events			
13	Mon Sep 14 00:54:48 2009	RCDAQ	Run 201 started
Run 201 started with file rcdaq-00000201-0000.evt			
12	**Sun Sep 13 19:24:19 2009**	**RCDAQ**	**Run 118 ended**
Run 118 ended with 11489 events			
11	Sun Sep 13 19:19:43 2009	RCDAQ	Run 118 started
Run 118 started with file rdcaq_0000000118_0000.evt			
10	Sat Sep 12 22:55:01 2009	RCDAQ	Run 109 ended
Run 109 ended with 23 events			
9	Sat Sep 12 22:54:29 2009	RCDAQ	Run 109 started
Run 109 started with file rdcaq_0000000109_0000.evt			

Wrapping up

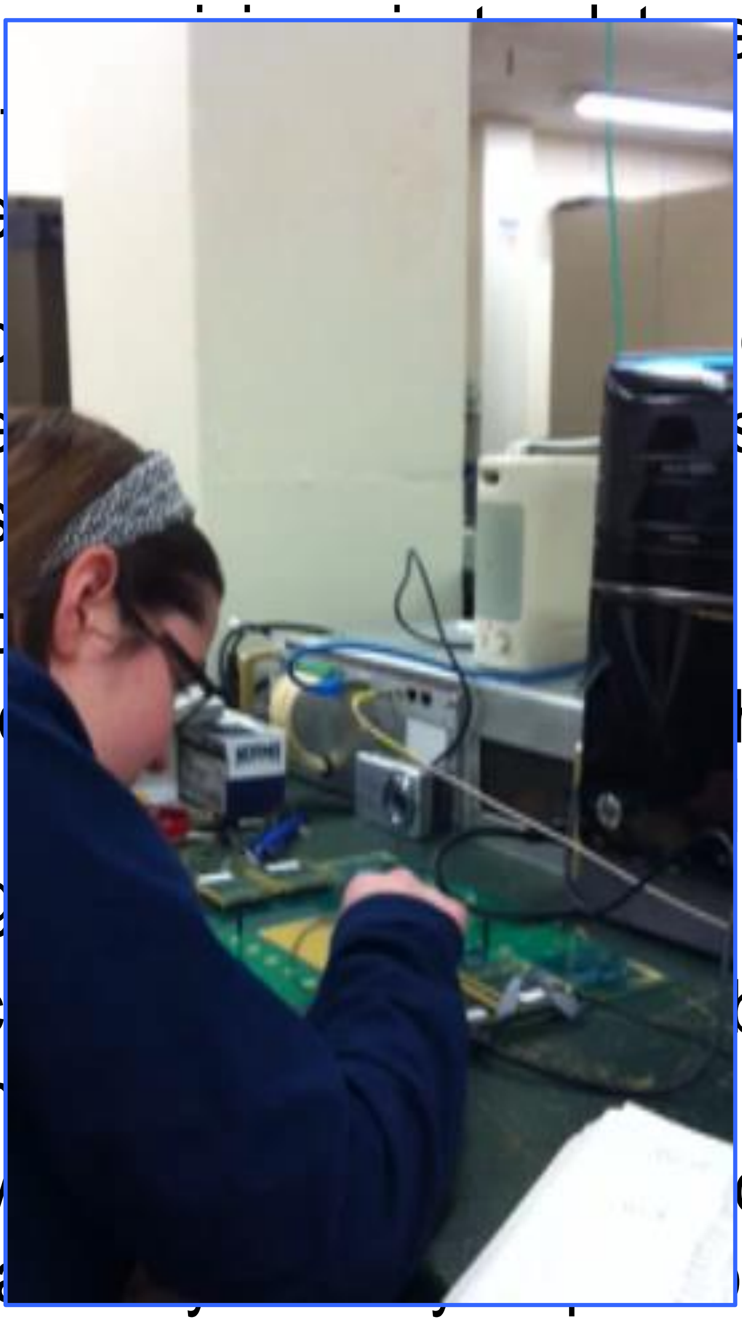
For th e harp
on a
she

Almo of a
ne s
as

A gro
th hey
a)
rig

Injec by
or
ey d
pl n

They came up with…



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
```

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.

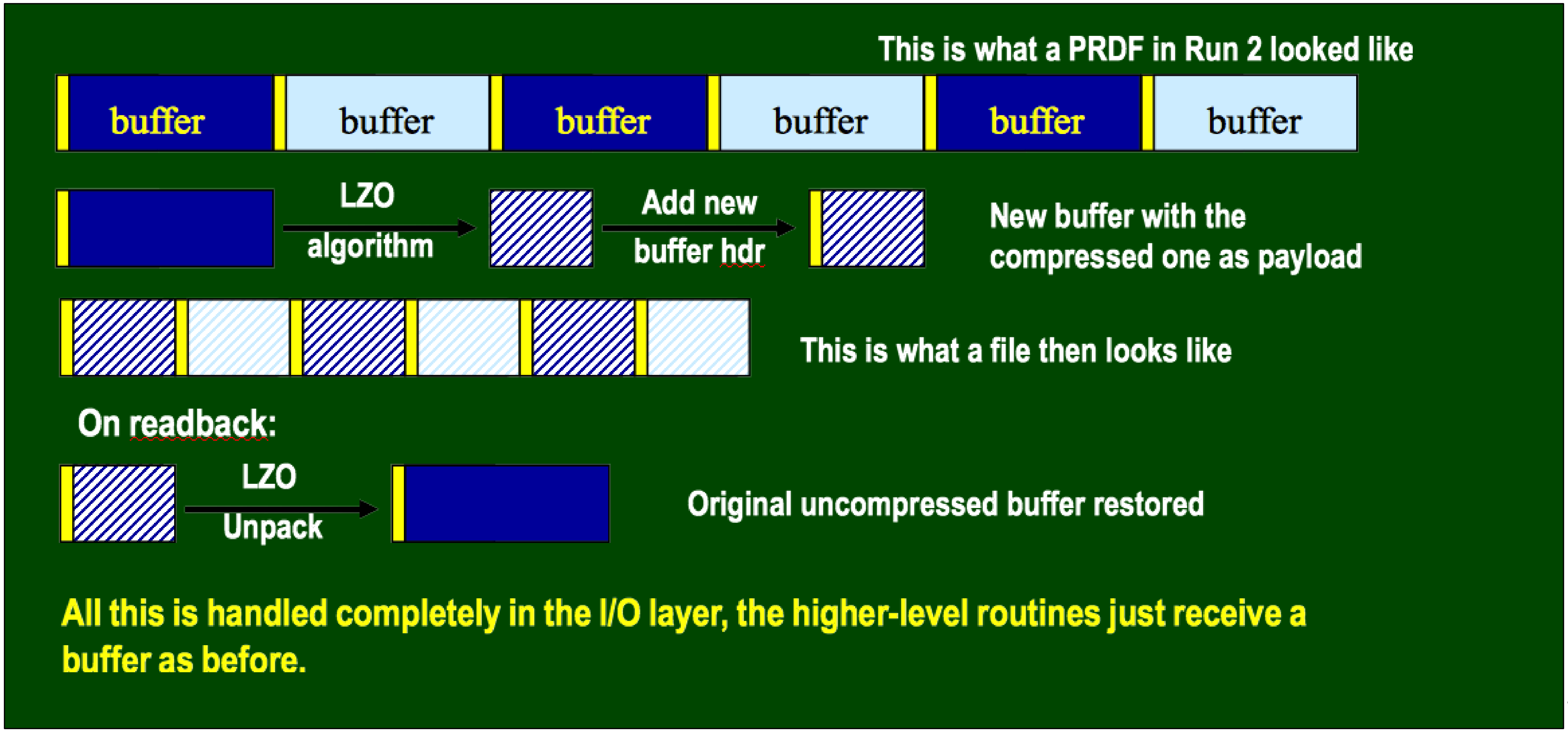
THANK YOU!

This page is intentionally left blank

Data Compression

We found that the raw data are still gzip-compressible after zero-suppression and other data reduction techniques

Early on, we had a compressed raw data format that supports a late-stage compression


This is what a PRDF in Run 2 looked like
buffer
buffer
buffer
buffer
buffer
buffer
LZO
algorithm
Add new
buffer hdr
New buffer with the compressed one as payload
This is what a file then looks like
On readback:
LZO
Unpack
Original uncompressed buffer restored

All this is handled completely in the I/O layer, the higher-level routines just receive a buffer as before.

A device_filenumber_delete example

You may have wondered what this is for…

Say you want to inject something into the datastream every 5 minutes or so. In this example, a temperature reading

If a file isn't there, no packet is generated

So we set up a cron job that reaches out to temperature-sensing board and creates a file "temperatures.dat" every 5 minutes.

In this way, we capture the file and numbers only in one event, then it's gone

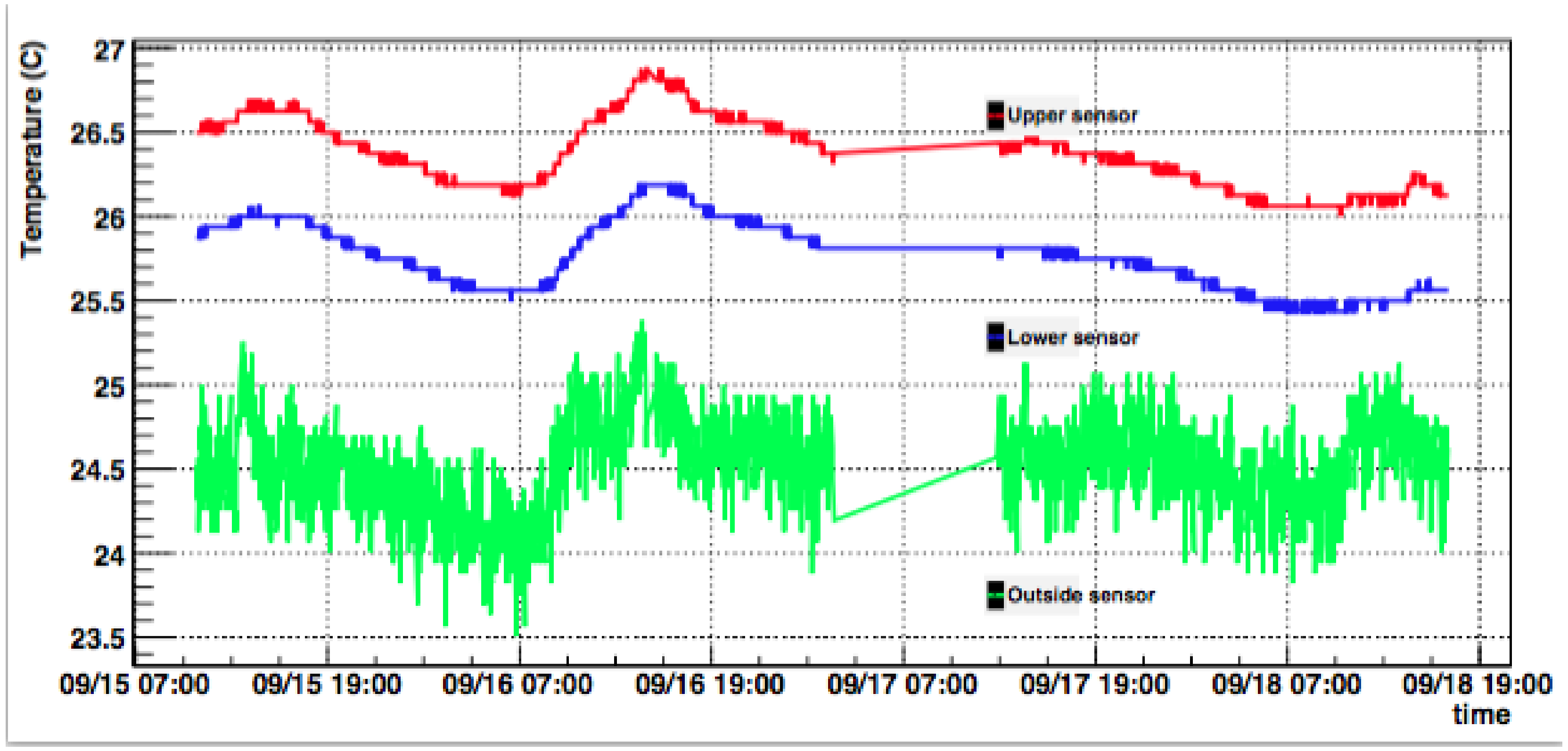
```
rcdaq_client create_device device_file 1 4001
  /home/hcal/drs_setup/temperatures.dat

rcdaq_client create_device device_filenumbers_delete 1 4002
  /home/hcal/drs_setup/temperatures.dat
```

This was a setup testing Silicon Photomultipliers, so the temperature stability is important

The Temperatures over time

```
$ ddump -O -n 1000 -p 4002 -g -d  \\
  /data/hcal/cosmics/cosmics_0000000115-0000.evt | \\
  grep '0 |' | awk '{print $4, $6,  $NF}'
25750 26312 24125
25750 26312 24250
25750 26312 23875
25750 26312 24187
25750 26312 23937
25750 26312 24312
25750 26312 24187
25687 26312 24000
25750 26312 24312
25750 26312 23875
25750 26312 24375
25687 26312 24125
25687 26312 24187
25750 26312 24062
25750 26312 24187
25687 26250 24187
25750 26312 24312
. . .
```


Temperature (C)
Upper sensor
Lower sensor
Outside sensor
09/15 07:00
09/15 19:00
09/16 07:00
09/16 19:00
09/17 07:00
09/17 19:00
09/18 07:00
09/18 19:00
time