Part 1:
RCDAQ and Run Control in sPHENIX

I will explain how our Run Control, in conjunction with our "GTU" (what we call GL1/GTM), manages all DAQ operations with the click of one (or maybe two) buttons

I want to remind us that any operation synchronized to a particular beam crossing cannot be accomplished via software controls but needs support from the GTU

I'll explain

- how Run Control manages the operations of many RCDAQs
- how our GTU (especially the "GTM" part of it) operates, and how it synchronizes things to a beam crossing
- how Run Control controls our GTU

Run Control does not *set up* any RCDAQ (or GTU) configurations. It merely manages the operations (start/stop etc) of already configured instances.

Time scales

I tend to explain how RCDAQ (and, by extension, Run Control) is inherently scriptable. And indeed, most *configurations* are performed with scripts (bash or python).

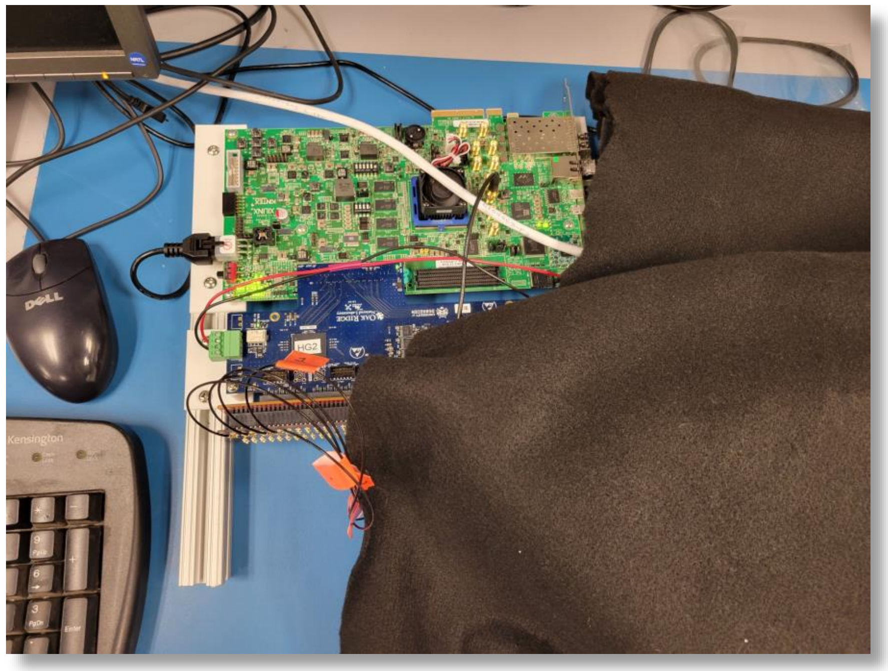
Run Control is for *operations*. This is a C++ client-server pair that takes care of that.

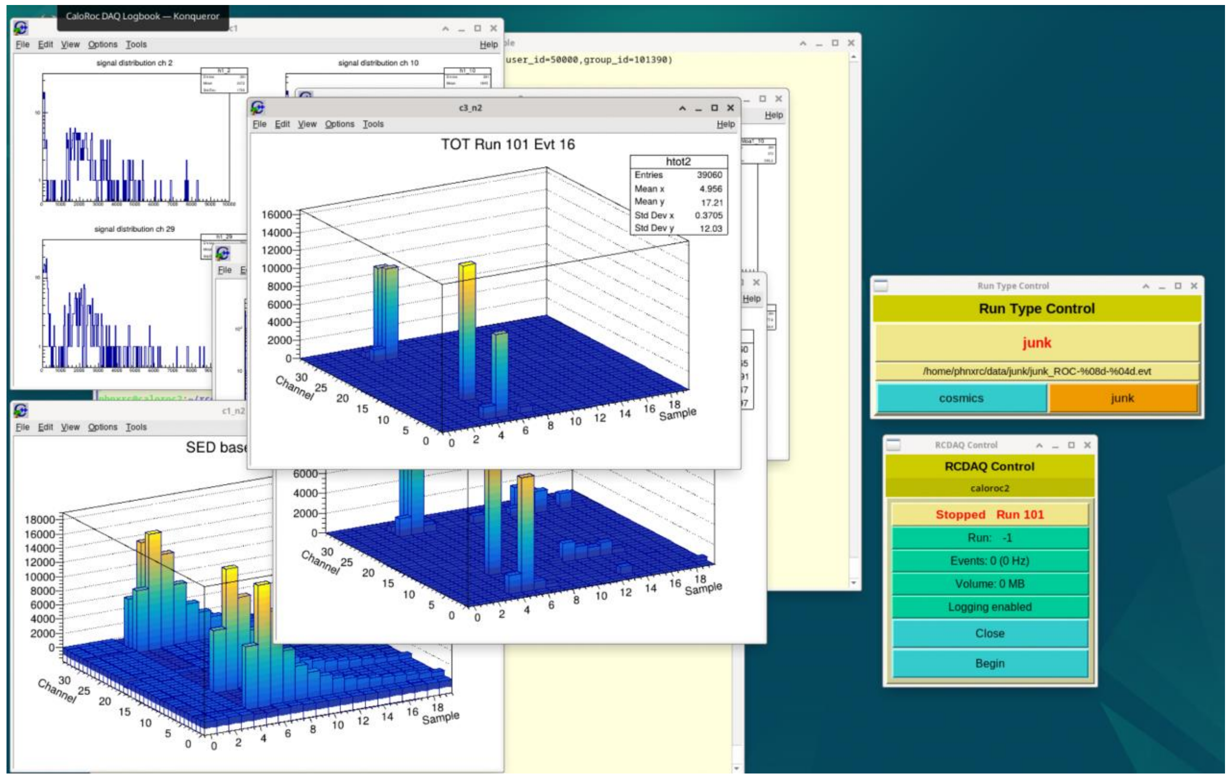
Let me break it down –

- Configuration: done every couple of days, whenever some changes happen (like during MD or APEX or so.) Colloquially: "Configure it on Monday, keep running through the week".
- Operation: With beam, our "runs" last typically an hour, sometimes less if some detector has an issue and needs attention
- Finally, the GTU is running crossing–aware with firmware on nanosecond time scales.

RCDAQ and Run Control in sPHENIX

RCDAQ can be used in a standalone manner – think of test beams, bench-top tests, that kind of thing. A good example is the currently ongoing development towards the BHCal prototype.





/home/phnxrc/data/junk/junk_ROC-%08d-%04d.evt

Stopped Run 101

Run: -1

Events: 0 (0 Hz)

Volume: 0 MB

Logging enabled

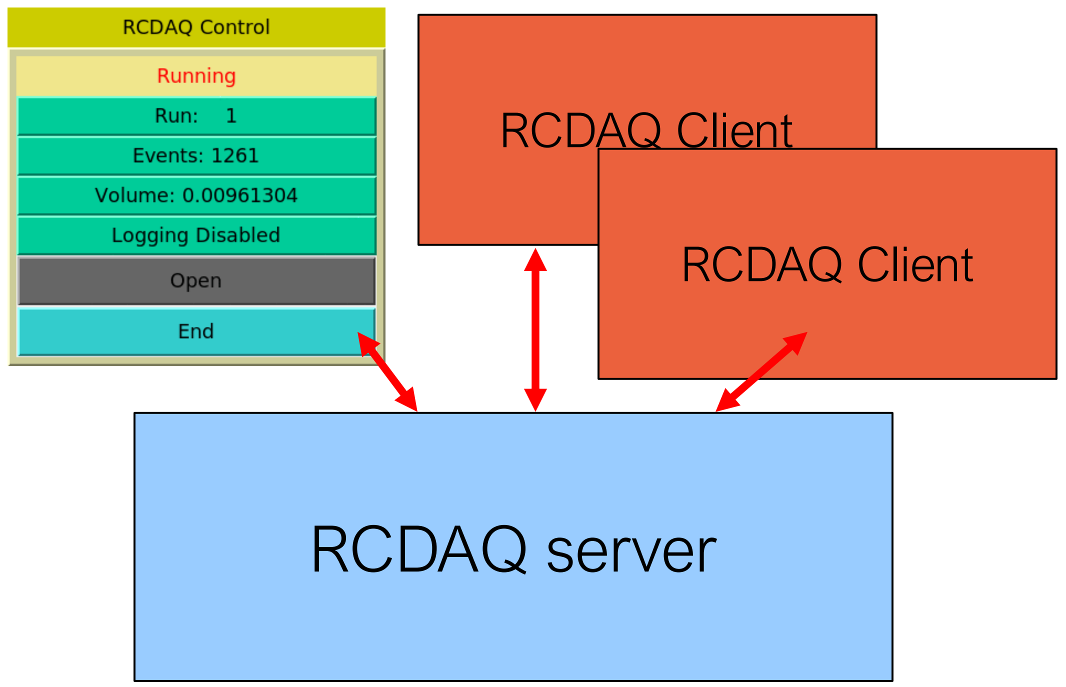
Here you have one RCDAQ instance reading out some kind of readout device – the H2GCROC3 in this case

What is Run Control?

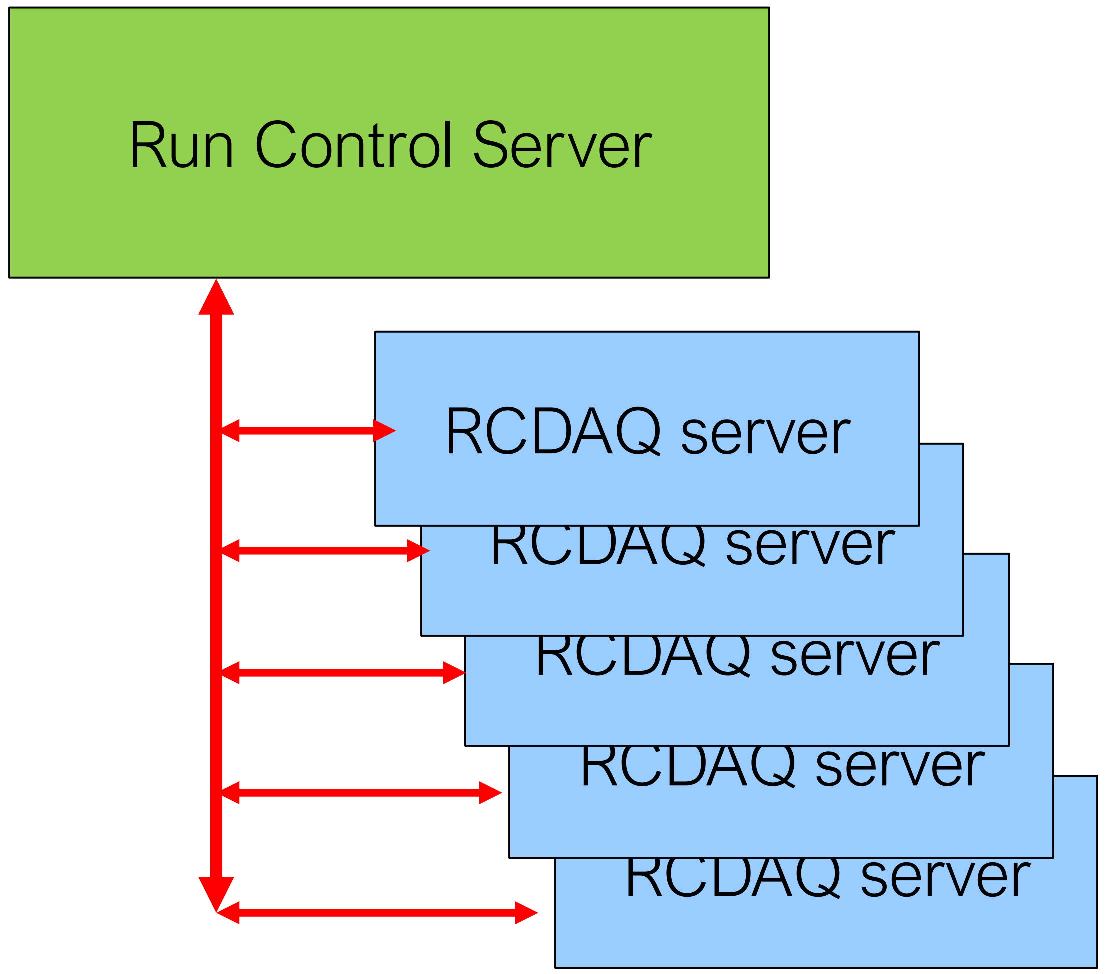
Similar to the rcdaq_server that is externally controlled via a communications protocol, the **rc_server** runs in the background and interfaces with many rcdaq_servers

It makes all RCDAQ instances take the same action, as instructed

Standalone



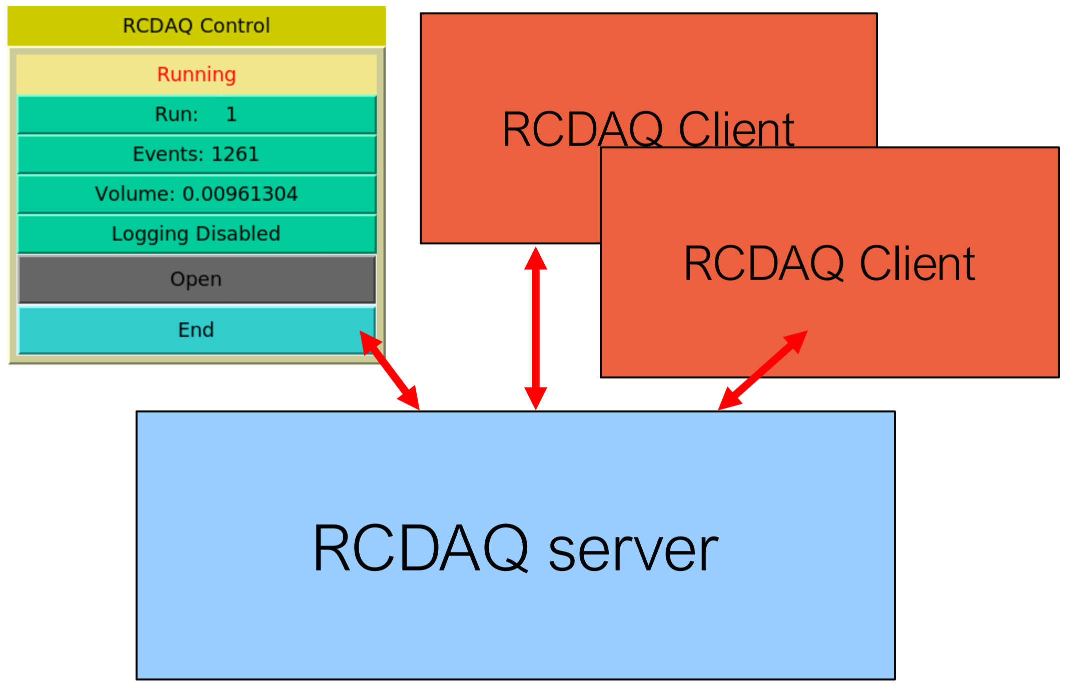
As a cohort



What controls Run Control?

Just like the rcdaq_server, the rc_server has its clients, too

rc_client controls every aspect of the rc_server



RCDAQ Control

Running

Run: 1

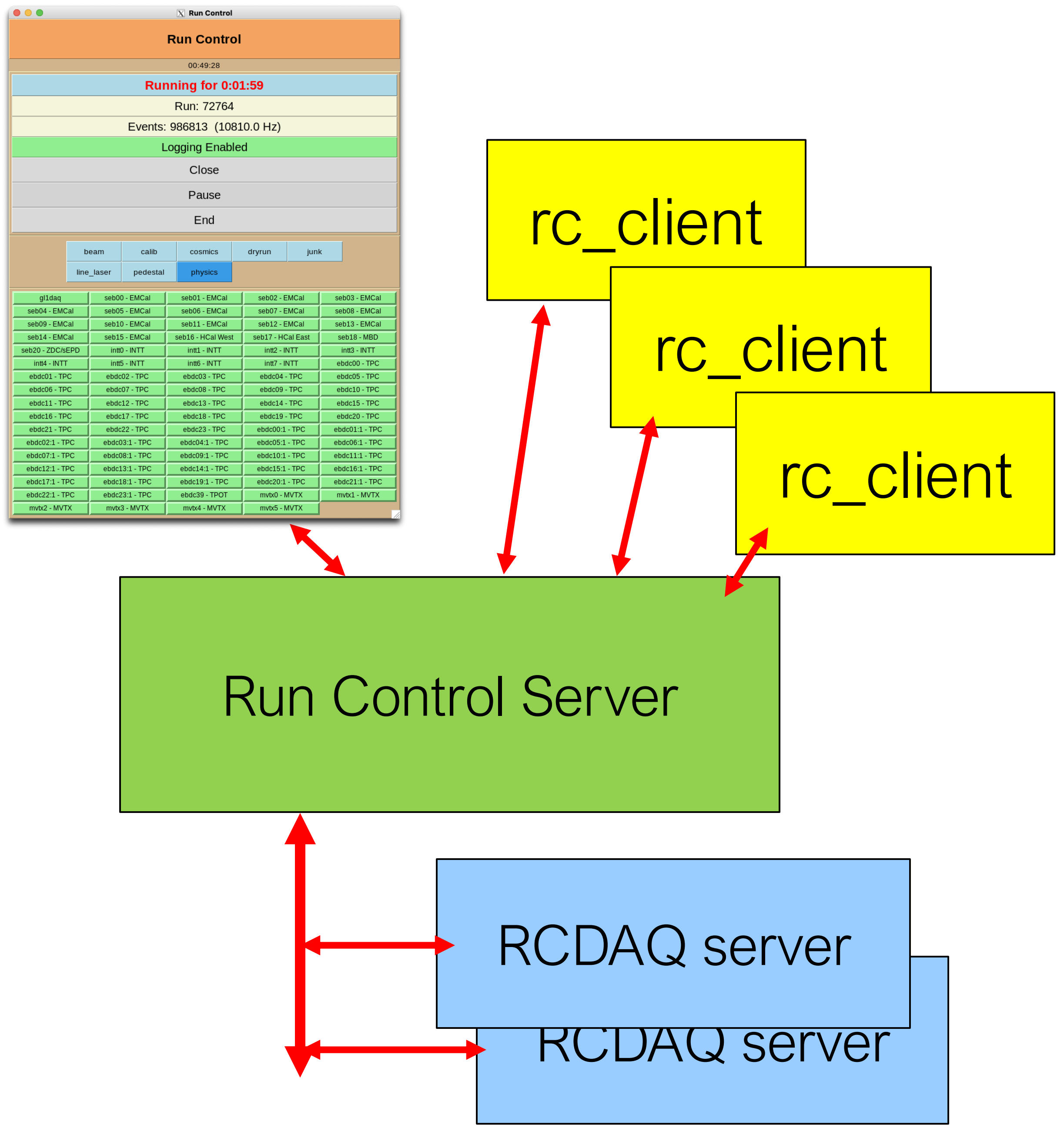
Events: 1261

Volume: 0.00961304

Logging Disabled

Open

End



Run Control

00:49:28

Running for 0:01:59

Run: 72764

Events: 986813 (10810.0 Hz)

Logging Enabled

gl1daq	seb00 - EMCal	seb01 - EMCal	seb02 - EMCal	seb03 - EMCal
seb04 - EMCal	seb05 - EMCal	seb06 - EMCal	seb07 - EMCal	seb08 - EMCal
seb09 - EMCal	seb10 - EMCal	seb11 - EMCal	seb12 - EMCal	seb13 - EMCal
seb14 - EMCal	seb15 - EMCal	seb16 - HCal West	seb17 - HCal East	seb18 - MBD
seb20 - ZDC/sEPD	intt0 - INTT	intt1 - INTT	intt2 - INTT	intt3 - INTT
intt4 - INTT	intt5 - INTT	intt6 - INTT	intt7 - INTT	ebdc00 - TPC
ebdc01 - TPC	ebdc02 - TPC	ebdc03 - TPC	ebdc04 - TPC	ebdc05 - TPC
ebdc06 - TPC	ebdc07 - TPC	ebdc08 - TPC	ebdc09 - TPC	ebdc10 - TPC
ebdc11 - TPC	ebdc12 - TPC	ebdc13 - TPC	ebdc14 - TPC	ebdc15 - TPC
ebdc16 - TPC	ebdc17 - TPC	ebdc18 - TPC	ebdc19 - TPC	ebdc20 - TPC
ebdc21 - TPC	ebdc22 - TPC	ebdc23 - TPC	ebdc00:1 - TPC	ebdc01:1 - TPC
ebdc02:1 - TPC	ebdc03:1 - TPC	ebdc04:1 - TPC	ebdc05:1 - TPC	ebdc06:1 - TPC
ebdc07:1 - TPC	ebdc08:1 - TPC	ebdc09:1 - TPC	ebdc10:1 - TPC	ebdc11:1 - TPC
ebdc12:1 - TPC	ebdc13:1 - TPC	ebdc14:1 - TPC	ebdc15:1 - TPC	ebdc16:1 - TPC
ebdc17:1 - TPC	ebdc18:1 - TPC	ebdc19:1 - TPC	ebdc20:1 - TPC	ebdc21:1 - TPC
ebdc22:1 - TPC	ebdc23:1 - TPC	ebdc39 - TPOT	mvtx0 - MVTX	mvtx1 - MVTX
mvtx2 - MVTX	mvtx3 - MVTX	mvtx4 - MVTX	mvtx5 - MVTX	

An

The r hereabouts and how to control them

And i

Here he 8/20 APEX day (tests w/o beam)

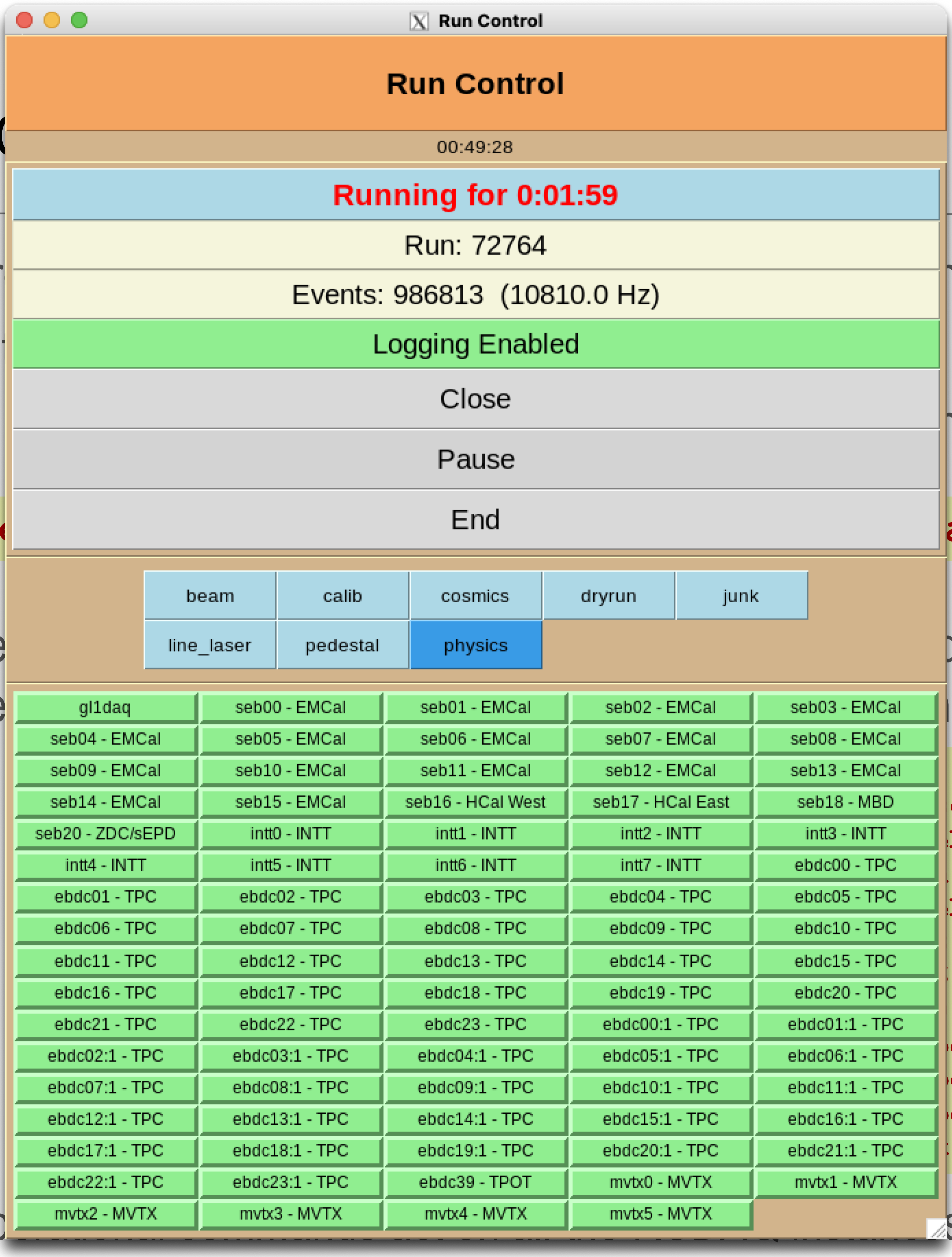
```
rc_se ations/hostlist.dat
```

rc_se d takes the list of constituents from a file e –f option is often easier.

Here

```
eb02 seb03 seb04 seb05 seb06 seb07
b11 seb12 seb13 seb14 seb15 seb16
tt0 intt1 intt2 intt3 intt4 intt5
bdc01 ebdc02 ebdc03 ebdc04 ebdc05
 ebdc09 ebdc10 ebdc11 ebdc12 ebdc13
 ebdc17 ebdc18 ebdc19 ebdc20 ebdc21
:1 ebdc01:1 ebdc02:1 ebdc03:1 ebdc04:1
dc07:1 ebdc08:1 ebdc09:1 ebdc10:1
dc13:1 ebdc14:1 ebdc15:1 ebdc16:1
dc19:1 ebdc20:1 ebdc21:1 ebdc22:1
0 mvtx1 mvtx2 mvtx3 mvtx4 mvtx5
```

All op s in the list.



Run Control

00:49:28

Running for 0:01:59

Run: 72764

Events: 986813 (10810.0 Hz)

Logging Enabled

Close

Pause

End

beam	calib	cosmics	dryrun	junk
line_laser	pedestal	physics		

gl1daq	seb00 - EMCal	seb01 - EMCal	seb02 - EMCal	seb03 - EMCal
seb04 - EMCal	seb05 - EMCal	seb06 - EMCal	seb07 - EMCal	seb08 - EMCal
seb09 - EMCal	seb10 - EMCal	seb11 - EMCal	seb12 - EMCal	seb13 - EMCal
seb14 - EMCal	seb15 - EMCal	seb16 - HCal West	seb17 - HCal East	seb18 - MBD
seb20 - ZDC/sEPD	intt0 - INTT	intt1 - INTT	intt2 - INTT	intt3 - INTT
intt4 - INTT	intt5 - INTT	intt6 - INTT	intt7 - INTT	ebdc00 - TPC
ebdc01 - TPC	ebdc02 - TPC	ebdc03 - TPC	ebdc04 - TPC	ebdc05 - TPC
ebdc06 - TPC	ebdc07 - TPC	ebdc08 - TPC	ebdc09 - TPC	ebdc10 - TPC
ebdc11 - TPC	ebdc12 - TPC	ebdc13 - TPC	ebdc14 - TPC	ebdc15 - TPC
ebdc16 - TPC	ebdc17 - TPC	ebdc18 - TPC	ebdc19 - TPC	ebdc20 - TPC
ebdc21 - TPC	ebdc22 - TPC	ebdc23 - TPC	ebdc00:1 - TPC	ebdc01:1 - TPC
ebdc02:1 - TPC	ebdc03:1 - TPC	ebdc04:1 - TPC	ebdc05:1 - TPC	ebdc06:1 - TPC
ebdc07:1 - TPC	ebdc08:1 - TPC	ebdc09:1 - TPC	ebdc10:1 - TPC	ebdc11:1 - TPC
ebdc12:1 - TPC	ebdc13:1 - TPC	ebdc14:1 - TPC	ebdc15:1 - TPC	ebdc16:1 - TPC
ebdc17:1 - TPC	ebdc18:1 - TPC	ebdc19:1 - TPC	ebdc20:1 - TPC	ebdc21:1 - TPC
ebdc22:1 - TPC	ebdc23:1 - TPC	ebdc39 - TPOT	mvtx0 - MVTX	mvtx1 - MVTX
mvtx2 - MVTX	mvtx3 - MVTX	mvtx4 - MVTX	mvtx5 - MVTX	

Let’s look at “rc_begin” (start a run)

These are the steps that happen:

All constituent RCDAQs are instructed to transition into data taking mode, in an *asynchronous* way (don’t wait for completion, return right away):

```
for ( it = HostList.begin(); it!= HostList.end(); ++it)
    {
          status |=  (*it)->rc_begin_immediate(TheRun, s);
     }
```

Then the rc_server makes a “sync” pass, wait for all RCDAQs to be ready (up to here, this is the equivalent of “ready – set” in a race). No data are flowing yet.

```
for ( it = HostList.begin(); it!= HostList.end(); ++it)
 {
    status |= (*it)->rc_sync(s);
 }
```

Finally, the rc_server issues a “start run” to the GTU. We’ll get there in a minute. (That’s the “start shot” in that race that makes data flow).

```
  GL1GTM->gtm_startrun(s);
```

What’s that asynchronous start of run and “sync”?

Each rcdaq takes a few seconds to get going, depending on how much meta-data we collect

Typically 3-5 seconds or so

If we’d do that sequentially for our 84 instances, we’d have > 250s startup time

That’s why we only issue the asynchronous “start run” command and return immediately

Now all RCDACs execute their start-run tasks concurrently

We then ask all of them if they are done. Here we have wait for the slowest contestant, but we wait no more than a few seconds

In this way we can start a run with minimal delay

Run Control Mode for RCDAQ

In a setup with a single RCDAQ instance (think test beam), that instance calls all the shots

Under run control, we reduce the autonomy of RCDAQ, so the rc_server is truly the boss

RCDAQ in Run Control Mode also reduces its interaction with the database (since RC is responsible for adding most information).

It only adds the info to the DB that no one else knows, like data volumes and such

```
 $ RCDAQHOST=seb00 daq_status -l
seb00 -  Stopped
  Filerule:     /bbox/bbox0/W/emcal/junk/junk_seb00-%08d-%04d.prdf
  Logging enabled
  Number of buffers/write threads: 94 Buffersize: 64MB
  compression enabled
  MD5 calculation enabled
  File rollover: 20GB
  Run Control Mode enabled
  . . .
```

Beam crossing-synchronous start of streaming

What I showed so far is the equivalent of the “ready – set” call in a race (but not “go”)

As I said, any beam crossing-synchronous start of the streaming data flow across many front-ends (DAMs/FELIXs) requires support from the GTU.

That’s then the “go”, or the start shot.

Per-crossing "mode bits"

Let me briefly explain the concept of our GTM "mode bits"

Those 8-bit codes allow us to control the "mode" of a given front-end component on a per-crossing basis

Example (was used in PHENIX) : every couple of seconds, in the abort gap, ask for an ADC readout of the Calos, and suspend zero-suppression for the data in that particular crossing. That gives you a periodic in-beam pedestal measurement.

You see why those codes are called "mode" bits - they can change the mode of the readout just for one crossing.

Each different front-end is free to implement its own modebits (within reason) and is free to ignore "foreign" modebits it doesn't recognize. Many codes are standardized.

Another example: Our calorimeters' controllers listen to the modebits, and recognize a "fire the LED!" code for calibration runs. (You need that LED pulse to fall in the abort gap)

The takeaway here: mode bits can trigger actions on a per-crossing basis.

Modebit management

RHIC has 120 bunch “buckets”/crossings – one full rotation is about 12us

Our GL1/GTM unit has 15 individual “vGTMs”, each detector system its own (we don’t have 15 detectors, there are “spares”)

The vGTMs know (from the v124) when “bunch 0” is passing by (and where the abort gap is)

Each vGTM has 16 “banks” with modebit information for 120 individual bunch crossings each

So each such bank covers a full RHIC rotation

The vast majority of modebits are empty (0) and have no special instructions

Better explained with an example…

Modebit banks

Here is the (dirt simple) setup for our calorimeters using 5 such 120-slot banks

What I show here is the modebit description language that gets converted into the banks' info

Each line has a bank number, a repeat count, a potential jump instruction, and a jump target

When we start, we issue a "run" modebit, like "prime yourself"

Later we issue a "reset" modebit, clearing all internal counters - now we are ready to go

```
#      repeat jump target values
 0        100    0    0
 1          0    0    0      9:RUN;
 2        100    0    0
 3          0    0    0      9:RESET;
 4     120000    1    4
```

0 – we make 100 turns doing nothing

1 – in crossing 9 we issue "run"

2 – 100 turns / 1.2ms of nothing

3 – crossing 9 sends "reset"

4 – 120,000 turns of nothing, followed by a jump back to itself, making an endless loop

What you see here is a one-shot init section in banks 0..3, followed by sending "nothing" indefinitely (until we say "stop")

Example: LED calibrations

Now it's easy to see how calibration events are triggered, and the readout is started.

We place the LED firing into the abort gap, where the detector is quiet

From there it's a calibrated propagation latency until we need to trigger the ADC readout (here: 29 crossings)

```
#         repeat jump target values
  0          100    0     0
  1            0    0     0      9:RUN
  2          100    0     0
  3            0    0     0      0:RESET
#  757 is about 10ms
  4          757    0     0
  5            0    0     0    119:PULSE
  6            0    1     4     28:FA
```

Lines 0…3 you already know

4 – we wait 757 turns - ~10ms

5 – "PULSE" fires the LEDs in crossing 119 (last bucket in abort gap)

6 – 29 beam crossings later we start the readout (FA = "Forced Accept")

… and then we jump back to 4, so we get ~100Hz of LED events.

… again until we say "stop"

And, at long last, the crossing-synchronous start

The streaming-readout detectors receive a “start streaming” (the “go” instructions) in a similar way (here: MVTX)

```
#      repeat jump target values
 0         10    0    0
 1          0    0    0      0: GTM_RESET; 100:FERST;
 2          0    0    0
 3          0    0    0      100: SOX;
 4     120000    1    4
```

Line 0: idle for 120 us

Line 1 – issue a “GTM_RESET” modebit in 0 (reset/re-lock w/GTM)

issue a ”FERST” (front-end reset), reset all counters at crossing 100

Line 2 – 12us idle time

Line 3 – “SOX” starts the streaming readout in crossing 100

Line 4 - the endless loop of “0” modebits that we already know

```
GL1GTM->gtm_startrun(s);
```

Summary

Run Control is a C++ client-server pair that can control many RCDAQ instances in a coherent manner

It streamlines the interaction with the RCDAQ instances and the GI1 (GTU)

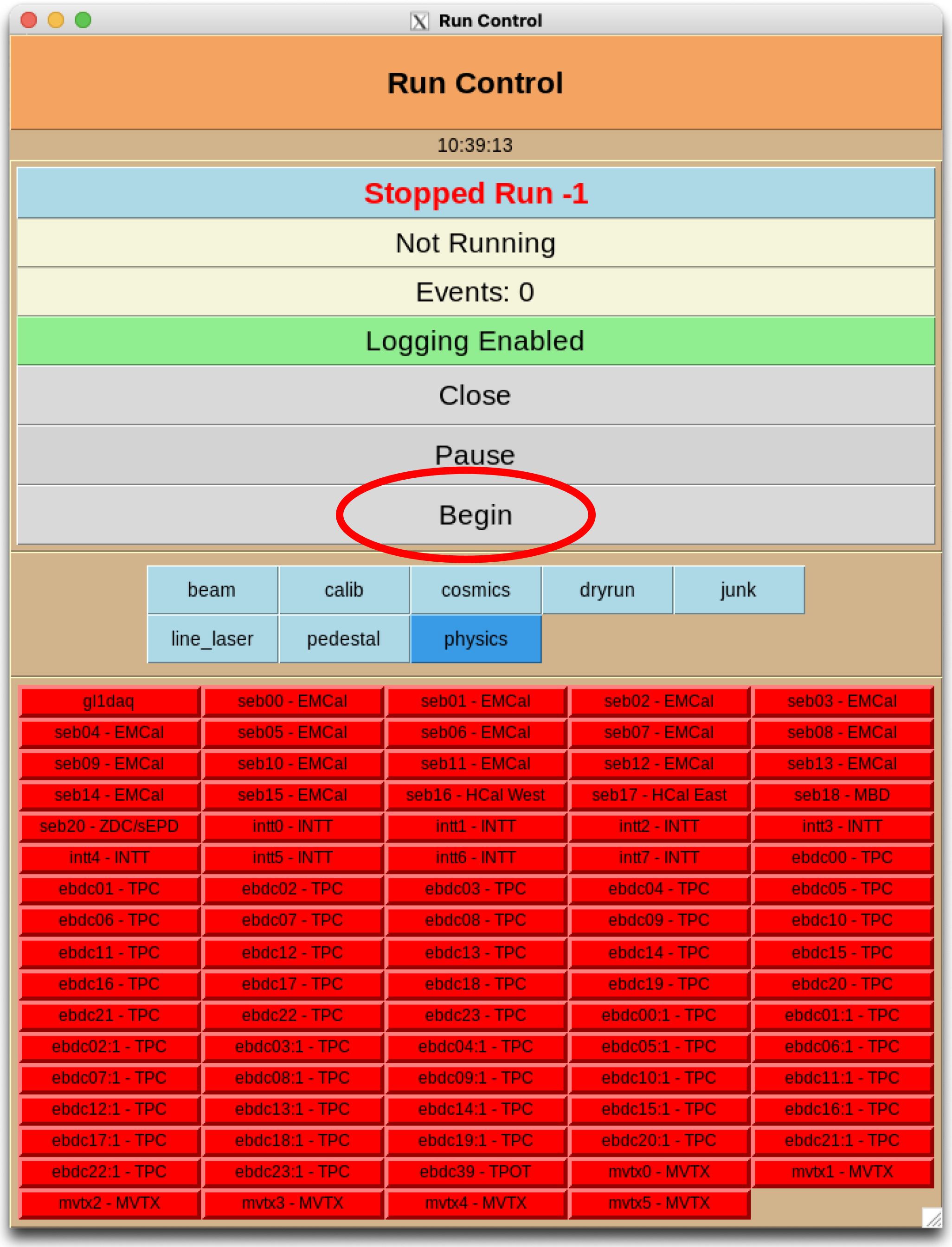
It has tons of features that I don't have time to get into –

- Controlled generation of run numbers (can only increase) – using an external "run number app"
- Hooks for additional begin/end run actions
- Generic information dissemination using MQTT (think database logging w/o the burden of DB library bindings)
- Dynamic constituent management (remove/add in case of problems)
- Special modes (like running w/o GTU for debugging etc etc etc)
- Automation via scripting as we know it from RCDAQ, where needed (e.g. calibration runs)
- **One-click operation for our shift crews**

One-Click operations

This is the main GUI that our DAQ operator normally sees. One click.

The GUI even has “skins” - a bored python-savvy operator on a no-beam shift ☺



10:39:13

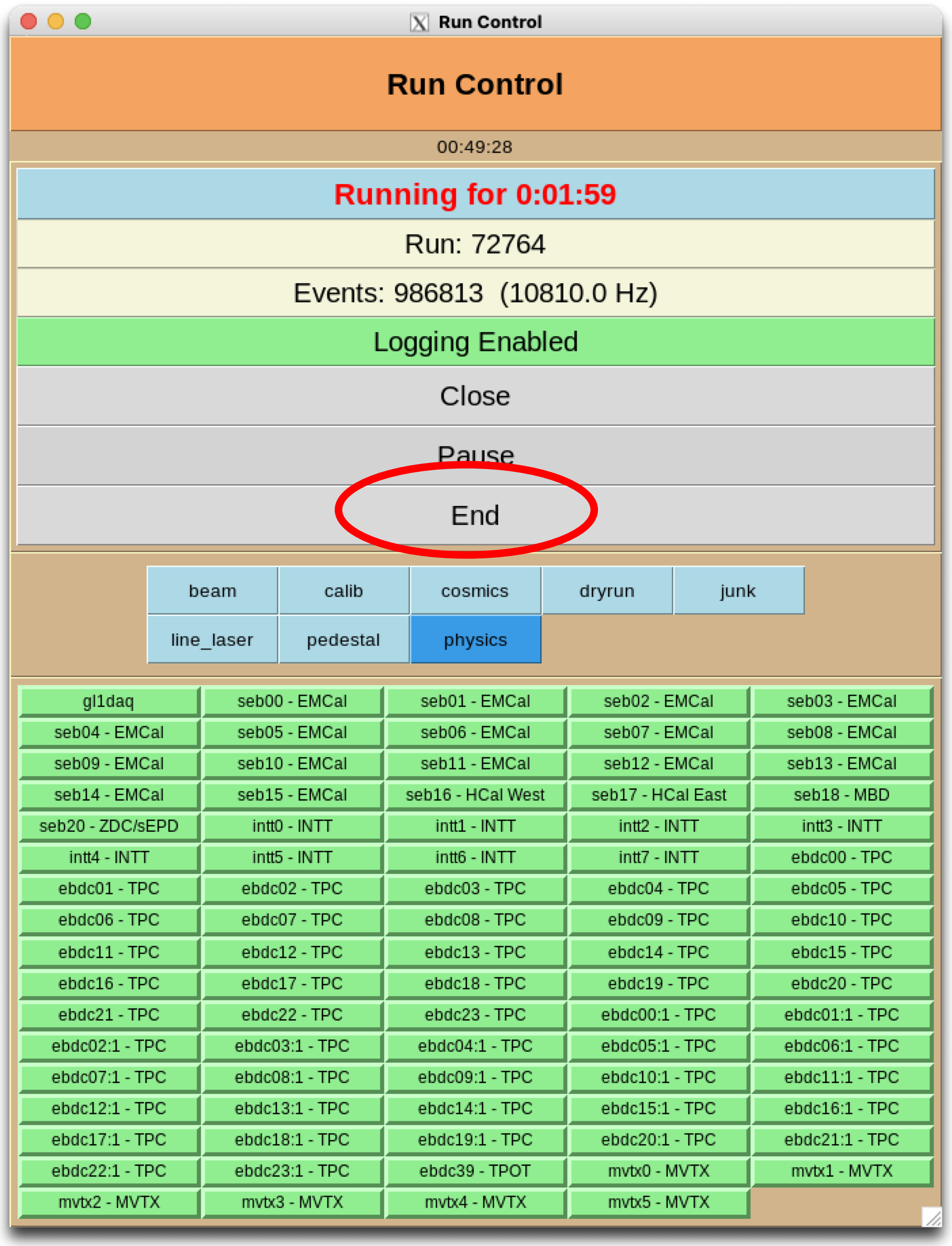
Stopped Run -1

Not Running

Events: 0

Logging Enabled

gl1daq	seb00 - EMCal	seb01 - EMCal	seb02 - EMCal	seb03 - EMCal
seb04 - EMCal	seb05 - EMCal	seb06 - EMCal	seb07 - EMCal	seb08 - EMCal
seb09 - EMCal	seb10 - EMCal	seb11 - EMCal	seb12 - EMCal	seb13 - EMCal
seb14 - EMCal	seb15 - EMCal	seb16 - HCal West	seb17 - HCal East	seb18 - MBD
seb20 - ZDC/sEPD	intt0 - INTT	intt1 - INTT	intt2 - INTT	intt3 - INTT
intt4 - INTT	intt5 - INTT	intt6 - INTT	intt7 - INTT	ebdc00 - TPC
ebdc01 - TPC	ebdc02 - TPC	ebdc03 - TPC	ebdc04 - TPC	ebdc05 - TPC
ebdc06 - TPC	ebdc07 - TPC	ebdc08 - TPC	ebdc09 - TPC	ebdc10 - TPC
ebdc11 - TPC	ebdc12 - TPC	ebdc13 - TPC	ebdc14 - TPC	ebdc15 - TPC
ebdc16 - TPC	ebdc17 - TPC	ebdc18 - TPC	ebdc19 - TPC	ebdc20 - TPC
ebdc21 - TPC	ebdc22 - TPC	ebdc23 - TPC	ebdc00:1 - TPC	ebdc01:1 - TPC
ebdc02:1 - TPC	ebdc03:1 - TPC	ebdc04:1 - TPC	ebdc05:1 - TPC	ebdc06:1 - TPC
ebdc07:1 - TPC	ebdc08:1 - TPC	ebdc09:1 - TPC	ebdc10:1 - TPC	ebdc11:1 - TPC
ebdc12:1 - TPC	ebdc13:1 - TPC	ebdc14:1 - TPC	ebdc15:1 - TPC	ebdc16:1 - TPC
ebdc17:1 - TPC	ebdc18:1 - TPC	ebdc19:1 - TPC	ebdc20:1 - TPC	ebdc21:1 - TPC
ebdc22:1 - TPC	ebdc23:1 - TPC	ebdc39 - TPOT	mvtx0 - MVTX	mvtx1 - MVTX
mvtx2 - MVTX	mvtx3 - MVTX	mvtx4 - MVTX	mvtx5 - MVTX	



00:49:28

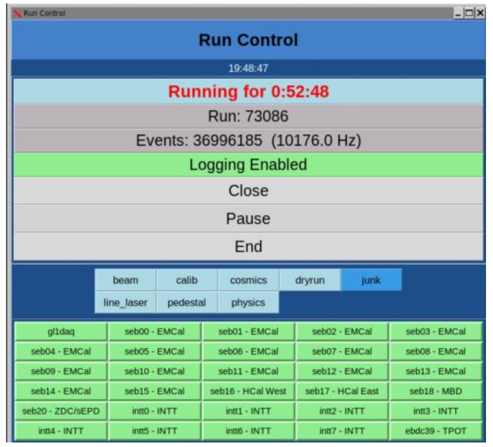
Running for 0:01:59

Run: 72764

Events: 986813 (10810.0 Hz)

Logging Enabled

gl1daq	seb00 - EMCal	seb01 - EMCal	seb02 - EMCal	seb03 - EMCal
seb04 - EMCal	seb05 - EMCal	seb06 - EMCal	seb07 - EMCal	seb08 - EMCal
seb09 - EMCal	seb10 - EMCal	seb11 - EMCal	seb12 - EMCal	seb13 - EMCal
seb14 - EMCal	seb15 - EMCal	seb16 - HCal West	seb17 - HCal East	seb18 - MBD
seb20 - ZDC/sEPD	intt0 - INTT	intt1 - INTT	intt2 - INTT	intt3 - INTT
intt4 - INTT	intt5 - INTT	intt6 - INTT	intt7 - INTT	ebdc00 - TPC
ebdc01 - TPC	ebdc02 - TPC	ebdc03 - TPC	ebdc04 - TPC	ebdc05 - TPC
ebdc06 - TPC	ebdc07 - TPC	ebdc08 - TPC	ebdc09 - TPC	ebdc10 - TPC
ebdc11 - TPC	ebdc12 - TPC	ebdc13 - TPC	ebdc14 - TPC	ebdc15 - TPC
ebdc16 - TPC	ebdc17 - TPC	ebdc18 - TPC	ebdc19 - TPC	ebdc20 - TPC
ebdc21 - TPC	ebdc22 - TPC	ebdc23 - TPC	ebdc00:1 - TPC	ebdc01:1 - TPC
ebdc02:1 - TPC	ebdc03:1 - TPC	ebdc04:1 - TPC	ebdc05:1 - TPC	ebdc06:1 - TPC
ebdc07:1 - TPC	ebdc08:1 - TPC	ebdc09:1 - TPC	ebdc10:1 - TPC	ebdc11:1 - TPC
ebdc12:1 - TPC	ebdc13:1 - TPC	ebdc14:1 - TPC	ebdc15:1 - TPC	ebdc16:1 - TPC
ebdc17:1 - TPC	ebdc18:1 - TPC	ebdc19:1 - TPC	ebdc20:1 - TPC	ebdc21:1 - TPC
ebdc22:1 - TPC	ebdc23:1 - TPC	ebdc39 - TPOT	mvtx0 - MVTX	mvtx1 - MVTX
mvtx2 - MVTX	mvtx3 - MVTX	mvtx4 - MVTX	mvtx5 - MVTX	



19:48:47

Running for 0:52:48

Run: 73086

Events: 36996185 (10176.0 Hz)

Logging Enabled

gl1daq	seb00 - EMCal	seb01 - EMCal	seb02 - EMCal	seb03 - EMCal
seb04 - EMCal	seb05 - EMCal	seb06 - EMCal	seb07 - EMCal	seb08 - EMCal
seb09 - EMCal	seb10 - EMCal	seb11 - EMCal	seb12 - EMCal	seb13 - EMCal
seb14 - EMCal	seb15 - EMCal	seb16 - HCal West	seb17 - HCal East	seb18 - MBD
seb20 - ZDC/sEPD	intt0 - INTT	intt1 - INTT	intt2 - INTT	intt3 - INTT
intt4 - INTT	intt5 - INTT	intt6 - INTT	intt7 - INTT	ebdc39 - TPOT