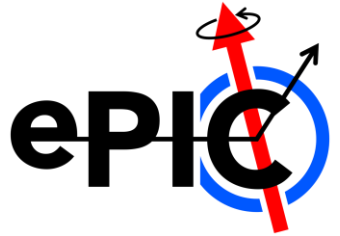


Part 3:

RCDAQ Plugins



I have explained that RCDAQ out of the box doesn't know how to read any particular device, minus a few built-in "pseudo-devices" (such as device_file, etc).

Every actual readout capability comes by way of a plugin that teaches RCDAQ how to read out the particular hardware (here: a DRS4 eval board)

```
$ daq_status -l
mlpvm5 - Stopped
  Filerule:      rcdaq-%08d-%04d.evt
  Logging disabled
  Buffer Sizes:   65536 KB adaptive
buffering: 15 s
  Web control Port: 8899
  Elog: not defined
-- defined Run Types: (none)
  No Plugins loaded
```

```
$ rcdaq_client load librcdaqplugin_drs.so
$ daq_status -l
mlpvm5 - Stopped
  Filerule:      rcdaq-%08d-%04d.evt
  Logging disabled
  Buffer Sizes:   65536 KB adaptive
buffering: 15 s
  Web control Port: 8899
  Elog: not defined
-- defined Run Types: (none)
  List of loaded Plugins:
    - DRS Plugin
```

RCDAQ Plugins

The `daq_status -l` (“long”) flags accumulate (`-l -l` or `-ll`).

With more verbosity, the plugin reports more details:

```
$ daq_status -ll
. . .
List of loaded Plugins:
- DRS Plugin, provides -
- device_drs (evtttype, subid, triggerchannel, triggerthreshold[mV], slope[n/p], delay[ns], speed,
start_ch, nch, baseline[mV]) - DRS4 Eval Board

- device_drs_by_serialnumber
      (evtttype, subid, serialnumber, triggerchannel, triggerthreshold[mV], slope[n/p],
delay[ns], speed, start_ch, nch, baseline[mV]) - DRS4 Eval Board
```

From this point on, we can make a “device_drs”, or a “device_drs_by_serialnumber”

(the drs is a USB device, so two devices might change their bus order, hence a more deterministic setup)

This is such a simple device that all configs can be made on the command line (that’s not normally the case)

```
rcdaq_client load librcdaqplugin_drs.so
rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 120 3
```

What's in a plugin?

On the outside, a plugin is a specially crafted shared library that registers itself with RCDAQ

This is no different from a shared library that we load into ROOT (here rootcint generates the “glue code”)

At the core is a class that inherits from a generic class “daq_device”

At the end of the day, there is a container class with all the daq_devices that were created

```
$ daq_list_readlist
File Device  Event Type: 9 Subevent id: 900 reading from /home/purschke/ISOTDAQ_rcdaq/setup.sh
DRS4 Eval Board  Event Type: 1 Subevent id: 1001 S/N 2662 Type 9 Trg 0x1 Thresh -150mV neg
delay 120 speed 3 (5GS) start 0 nch 1024 *Trigger*
```

What's that “event type”? It sets up when the device is to be read out – most events are type 1 (“data events”, or 2 (“streaming events”).

9 – begin-run event (you see above that we are preserving our setup script itself in the begin-run event for documentation purposes)

12 - end-run event

(some more)

The daq_device class

This is the “meat” of the readout.

The new class inherits from the daq_device class, and can implement the following methods:

```
class daq_device {  
  
public:  
  
    daq_device();  
    virtual ~daq_device();  
    virtual void identify(std::ostream& os = std::cout) const = 0;  
    virtual int max_length(const int etype) const = 0;  
    // functions to do the work  
    virtual int init(){return 0;};  
    virtual int endrun(){return 0;};  
    virtual int rearm( const int etype){return 0;};  
    virtual int put_data(const int, int *, const int) = 0;
```

Constructor and destructor do general initialization and tear-down

init() does initializations when a run begins, and endrun() is called at the end of the run

rearm() does whatever is needed to prepare the device to read new data

max_length() returns the maximum amount of words that device can deliver

identify(...) makes the device “identify” itself with a one-liner text

max_length()

Brief detour:

RCDAQ negotiates a maximum event size (per event type).

The events are almost always *much* smaller than that (zero-suppression, etc), but we need to know for the buffer management

We we start reading our devices, we cannot yet know their cumulative size (but we know the max!)

We are filling the current buffer. If a potential max-sized event will still fit into that buffer, we go ahead and put the data there

If the space left in the buffer is too small, we wrap up the buffer, submit it for “disposal”, and start a new one. The new data end up in the new buffer

We then inform RCDAQ/the buffer how much space we *actually* used, and it updates its parameters

Sometimes the max_length() is just a fixed value. Most often it is calculated based on the particular configuration of the device (like how many channels are being read out, etc)

Here is a very simple – fixed – one for the DRS4 eval board:

```
int daq_device_drs::max_length(const int etype) const
{
    if (etype != m_eventType) return 0;
    return (5*1024 + SEVTHEADERLENGTH +10 );
}
```

max_length() and event types

Look at this from before:

```
$ daq_list_readlist
File Device  Event Type: 9 Subevent id: 900 reading from /home/purschke/ISOTDAQ_rcdaq/setup.sh
DRS4 Eval Board  Event Type: 1 Subevent id: 1001 S/N 2662 Type 9 Trg 0x1 Thresh -150mV neg
delay 120 speed 3 (5GS) start 0 nch 1024 *Trigger*
```

```
int daq_device_drs::max_length(const int etype) const
{
    if (etype != m_eventType) return 0;
    return (5*1024 + SEVTHEADERLENGTH +10 );
}
```

This makes our device report 5K+change only for its event type (1 = data event), 0 for all others

Others can be really complex – our calorimeters read out as multiple groups of multiple ADC boards, each group can have 64, 128, or 192 channels

Each ADC board can provide a different number of samples per channel

All that influences the max number of words

Finally: put_data(...)

This is the call-back function that actually puts the data into a buffer.

It gets the event type, an address where to dump the data to, and a max amount of data it's allowed to write there:

```
int daq_device_drs::put_data(const int etype, int * adr, const int length )
{
    // etype    - the current event type being read
    // adr       - where we can write our data to
    // length    - the max number of words we are allowed to write
```

The last parameter is just a safety belt; in principle it comes directly from the earlier size negotiation

The daq_devices generally fall into a few different classes

- PCI devices, such as our FELIXes/DAMs
- Network devices, the SRS, our GL1/GTM, and the current ePIC H2GCROC3 readout
- USB devices (such as the DRS4 Eval board)

The network devices are close relatives; I used the SRS as basis to make the GL1, and that to make the H2GCROC3 readout

Let's look at our FELIX readout

We (that is, JohnK who programmed all the firmware, and I) made it so that the DAM shows up as a POSIX-compliant file descriptor

You can use `read(...)`, `select(...)`, use `ioctl`s, etc etc

That makes the readout super-easy:

https://github.com/sPHENIX-Collaboration/tpc_dam

```
int  daq_device_dam::init()
{
    pl_open(&_dam_fd, m_deviceName.c_str()); // just a wrapped open(...)
    . . .
}
```

```
int daq_device_dam::put_data(const int etype, int * adr, const int length )
{
    // some lines removed, a bit abridged, here is the main part:

    subevtdata_ptr sevt = (subevtdata_ptr) &adr; // map our packet structure on adr
    uint8_t *dest = (uint8_t *) &sevt->data; // and dest is now where our payload goes
    ssize_t  c = read(_dam_fd, dest, _length); // read payload

    // some wrapping up...
```


How does the “plugin” part work?

A plugin class inherits from the “RCDAQPlugin” class and has just 2 special member functions:

```
class dam_plugin : public RCDAQPlugin {  
  
    public:  
        int  create_device(deviceblock *db) ;  
        void identify(std::ostream& os = std::cout, const int flag=0) const;  
};
```

And inherits the constructor/destructor

```
class RCDAQPlugin  
{  
    public:  
        RCDAQPlugin()  
        {  
            plugin_register(this) ;  
        }  
        virtual ~RCDAQPlugin()  
        {  
            plugin_unregister(this) ;  
        }  
};
```

So when such a plugin object gets made, it registers itself with rcdaq

Loading plugins and making devices

This is what happens in one of the EBDCs (in real life it's a bit more elegantly scripted than this)

```
rcdaq_client load librcdaqplugin_dam.so
rcdaq_client create_device device_dam 1 4000 1 64 1 /dev/dam0
```

At this point, RCDAQ has one plugin

```
List of loaded Plugins:
- DAM Plugin, provides -
- device_dam (evttype, subid [, npackets, trigger] ) - DAM FELIX Board
```

When we ask for a device "device_dam", RCDAQ first looks if itself knows such a device (like "device_file")

If not, as for "device_dam", it traverses the plugins and asks, "do you know that?"

If yes, then the plugin makes the device, registers it, done, returns "ok".

Another return code means "no, not mine"

And yet another means, "mine indeed, but there was an issue with the parameters" (error):



```
$ rcdaq_client create_device device_drs -- 1
Device needs at least 2 parameters

$ rcdaq_client create_device device_drx -- 1 1001 0x21 -150 negative 120 3
Unknown device
```

And that's pretty much it

The rcdaq distribution comes with an example plugin that can be used as a basis to make your own

<https://github.com/sPHENIX-Collaboration/rcdaq>

 example_plugin.cc	more steps tpward plugins	14 years ago
 example_plugin.h	more steps tpward plugins	14 years ago

By now we have so many plugins that I usually take an existing, similar one (like for the H2GCROC3 one) and make a few changes - compare those two

https://git.racf.bnl.gov/gitea/sPHENIX/GL1_readout and

https://git.racf.bnl.gov/gitea/EIC/EIC_H2GCROC3

That H2GCROC3 probably took 3 hours including polishing

List of loaded Plugins:

- H2GCROC3 Plugin, provides -
- `device_h2gcroc3 (evtttype, subid, IP addr, trigger, nr_packets)` - readout an H2GCROC3

Just for completeness...

All the examples so far had just one plugin loaded. But we often (especially at test beams and lab tests) read more than one device.

Here is an older setup from an EIC FermiLab test beam where we read out a SRS and a DRS4 together (a GEM detector with the SRS and the FTBF Cherenkovs with the DRS4):

```
rcdaq_client load librcdaqplugin_srs.so
rcdaq_client load librcdaqplugin_drs.so
#...
rcdaq_client create_device device_drs -- 1 1020 0x10 -200 negative 400 1 0 1024
rcdaq_client create_device device_srs 1 1010 10.0.0.2 1
```

So, an arbitrary number of plugins can co-exist if needed, each bringing in its devices.

I don't have a SRS here, but I can build and load the plugin:

```
List of loaded Plugins:
- DRS Plugin, provides -
- device_drs (evtttype, subid, triggerchannel, triggerthreshold[mV], slope[n/p],
delay[ns], speed, start_ch, nch, baseline[mV]) - DRS4 Eval Board
- device_drs_by_serialnumber
(evttype, subid, serialnumber, triggerchannel, triggerthreshold[mV],
slope[n/p], delay[ns], speed, start_ch, nch, baseline[mV]) - DRS4 Eval Board
- SRS Plugin, provides -
- device_srs (evtttype, subid, IP addr) - readout an SRS crate
```

Summary -

The plugin concept is what makes RCDAQ so versatile

It would not be possible to build a monolithic binary with all required libraries for dozens of devices

While over time we have made dozens of plugins, sPHENIX has only 4 really dedicated ones (TPC/DAM, INTT, MVTX, Calorimeters, GL1)

Plugins for most commercial devices you are likely to encounter exist (DRS4, CAEN1742, Struck 3000, SRS, even the Saclay Dream readout, ...)

Of most recently made plugins for sPHENIX, the one for the DAM was by far the easiest

Followed by our GL1/GTM readout