PDAQ
A data acquisition system for lab tests of the PHENIX detectors

ONCS

Sept 3, 1997 – version 1.04

Contents

1 What is PDAQ

In 1997, PHENIX had several test beams and other test setups scheduled, which all needed a data acquisition system to read out or debug the detectors or some front-end electronics. The ONCS group in PHENIX decided to take advantage of the resulting call for support by designing a small system which uses the technologies and tools PHENIX intends to use (most notably the choice of the object-oriented programming language, `C++`, CORBA, and the Shlaer-Mellor methodology). In this way we would be able to support the detector groups and at the same time gain experience with the new technologies in a real-world setting, of course with the goal of making the software general enough that components of the small system (nick-named "`Pdaq`") could be re-used in the real big data acquisition system later.

With the exception of a few lines of assembly code, the system is written in `C++` using an object-oriented design, and the higher-level parts use the Shlaer-Mellor CASE tools to design and automatically generate the code.

The design of this system targets a small, fictitious setup with a few detectors and in the order of a thousand readout channels in VME or CAMAC, with full support for a test beam setup at the AGS. The system is highly configurable, and it was felt that the requirements of a typical setup at the AGS with a spill-structured beam, multiple hardware triggers, and high event rate would set a reasonable design goal and would be more than adequate for smaller setups where those features are not essential.

Because the use of the commercial CORBA software as one of the communications mechanisms restricts the system to those platforms where an implementation is available, we also support a lightweight but fully functional version of PDAQ which does not use CORBA at all.

The parts of the system are the software running in the front-end VME-processors with VxWorks as their operating system, the control and analysis support software running on a UNIX platform, and a seperate online monitoring package called `Pstuff`, which is described in a different manual.

As the time of this writing, this system has been used in 3 major projects:

- The Time Expansion Chamber (TEC) subsystem has used the Pdaq system first to debug the electronics and then to test the chambers at the AGS test beam;
- The lead glass detector which has been shipped from CERN used the system to verify the proper functioning of all detector modules, and performed a rough adjustment of the high voltage settings;
- the magnet mapping uses the system to read out the Hall probes measuring the magnetic field.

2 Modes of PDAQ operation

We said in the beginning that we support a lightweight version of the system which does not use CORBA at all, and runs of a large number of different UNIX platforms, including, most notably, LINUX.

The most direct way to run the system is called "VxWorks shell mode", because you use a terminal connected to the VxWorks processor or a UNIX window to log into the processor to give commands. This mode is the one which was used in the TEC beam test at the AGS and allows a very direct interaction with the system. Even if the system is operated in one of the other modes, some debug-style commands are available only from the VxWorks shell.

In many cases, this VxWorks mode turns out to be too inconvenient for automated measurements such as the lead glass high voltage adjustment procedure mentioned above. In order to automate such a task, the information from the online analysis is used to steer the data acquisition, which is not possible in VxWorks mode because there is no real connection between the analysis on the UNIX machine and the commands given to the VxWorks shell. For this reason, at the specific request of the EmCal group, we implemented what we call the "Unix shell mode". This mode does not add additional functionality to the system, but it makes the VxWorks shell commands used to control PDAQ available from the UNIX shell, so that there can be one UNIX process which controls the whole system. In addition to the Unix shell commands, there is also an application programming interface (API) available so that the data acquisition can be controlled from a program.

The highest form of PDAQ control is called "PHLOPS" mode, called after the "PHENIX Online Operating system". You can think of this system as a very comfortable remote control which adds a lot of functionality to the system on its own. This system is still work in progress, and due to the many changes to come, it will be described in a future version of this manual. For the time being, the VxWorks and UNIX shell modes are certainly powerful enough to support all lab tests scheduled for the PHENIX detectors.

3 Setting up a PDAQ system

3.1 A typical system

A typical PDAQ system is shown in figure 1 and consists of a VME crate with a VxWorks processor, a CAMAC crate, a UNIX system to receive the data, and the actual readout electronics, either in CAMAC or VME. Since all setups so far needed some CAMAC modules such as ADC's and TDC's, the system in its current form must have a CAMAC crate to support the trigger and busy logic, but support for different (CAMAC-less) hardware can

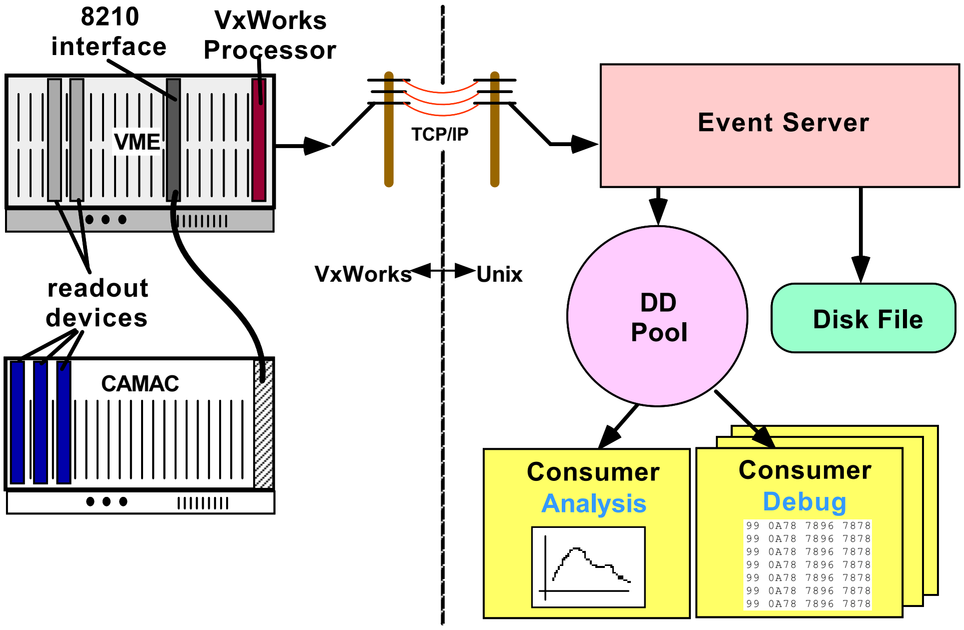

Figure 1: A typical PDAQ system (operating from the VxWorks shell)

be added as soon as more PHENIX-specific electronics becomes available. Right now the system relies on a CES 8210 VME-to-CAMAC interface to interrupt the readout processor and an output register in CAMAC crate 1 to reset the busy latch.

The readout processor can be either a Motorola MVME 162 or 167 (68040-based), or a MV2301 (PowerPC-based). When the processor receives an interrupt, it starts to read the devices which have been added to the readout list, and transfers the data through a TCP/IP connection to a UNIX host. There the data are written to disk and are put into a "data distributor" pool, from which monitoring processes can retrieve the event data.

3.2 A standard hardware triggerbusy logic

The data acquisition system usually needs a *trigger* which tells the processor to go and read out the devices in a list. A trigger is generated by some external trigger logic, for example, from a coincidence between the signals of scintillator paddles for a cosmic ray measurement. This trigger logic is of course highly setup-specific, but for the busy logic a standard setup has evolved over time. You will need a LeCroy 2323A Dual Gate generator to serve as a busy latch (this module is also available as a NIM version LeCroy 222), and a CAMAC output register (figure 2).

Set one of the gate generators to "indefinite", so once triggered, it will stay on forever. This will become our busy latch. Its `NIM` output connects to the `INT4` input of the 8210, which will trigger an interrupt in the readout processor. After finishing the readout, the processor will reset the busy by a pulse on output 2 of the output register (which must be in crate 1, slot 1).

Each time there is a pulse from the external trigger logic, and the busy is not on, we will trigger an event.

4 Basic operation

Once the system and the trigger logic has been set up, the user configures the system by certain commands either from the VxWorks shell or from the UNIX shell. For the examples, we will first show the VxWorks shell commands, and later show how to use the UNIX commands. The VxWorks prompt is usually "`->`", so we show it here to indicate that we type a command in the VxWorks shell.

In virtually any setup, the configuration commands will reside in script files which can later be executed automatically at boot-up, but for now we give them from the shell.

Most of the configurable parameters have reasonable defaults set up wherever possible, so you need to configure only those parameters which you actually want to change.

4.1 Initialisation

To start and initialize the whole system, give the command

```
-> daq_init
```

This command starts up all necessary processes, installs the interrupt handlers, creates objects, and so on. This must be the first command given.

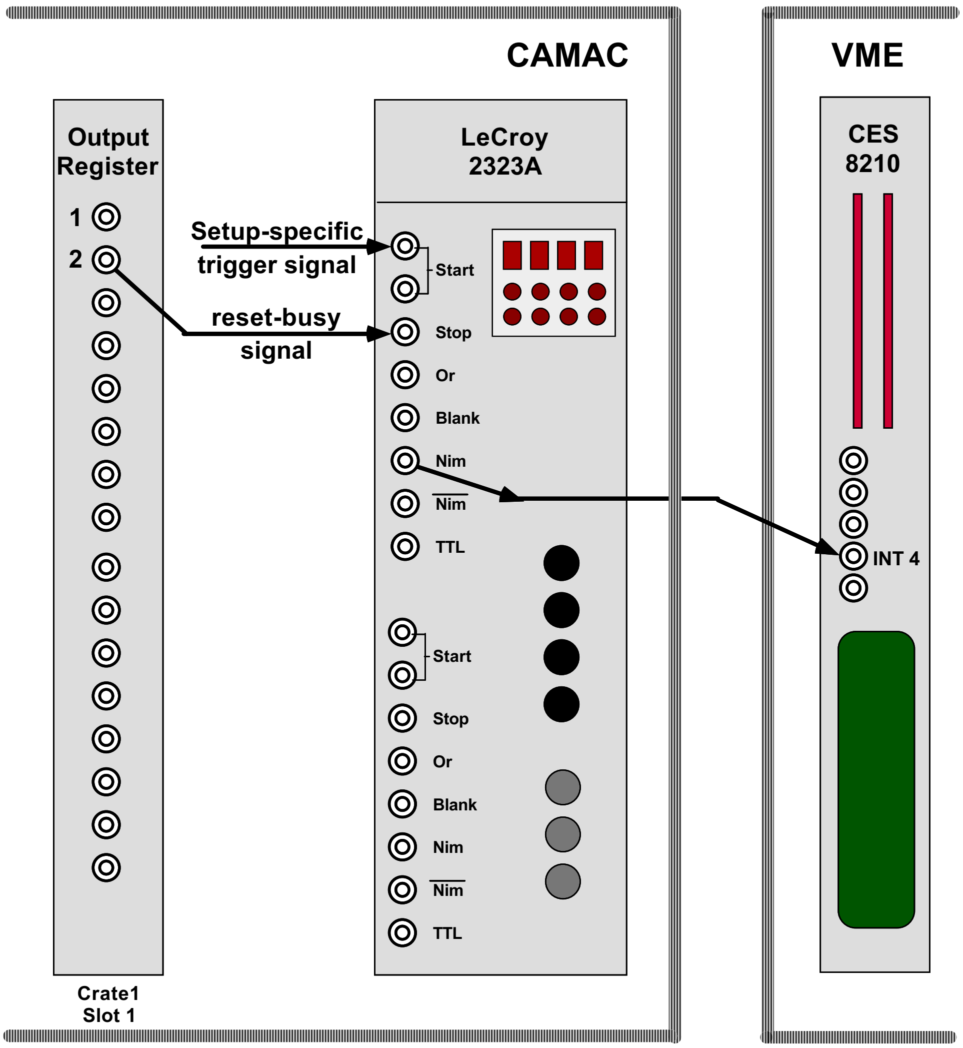

Figure 2: A standard trigger setup with a LeCroy 2323A module. The setup-specific trigger signal sets the gate (the busy) on the 2323A (which must be set to "indefinite" timing. The output of the 2323A triggers the interrupt through the INT4 input at the 8210. When the processor has finished reading the event, it will give a pulse on output 2 of the output register, and clear the busy.

4.2 Exercise the "empty" system

Now, without doing anything else, we can already exercise this minimal system. It will not read anything, but it will react to external triggers and reset the trigger logic. In fact, if you are setting up a system, this step is highly recommended in order to see if all hardware and software components have been setup properly. It is a good way to see if there is already a problem with the basic setup (rather than with any readout devices). Just hook up a NIM pulser to the "Start" input of the 2323A (figure 2), and type

```
-> daq_begin 1
```

This will start run number 1, and events will be triggered, and you should see the LED at the 2323A flicker. If this test works, the basic setup is ok. Type

```
-> daq_end
```

and the data taking will stop.

4.3 Defining readout devices

Each readout device is registered with the system (entered into the readout list) with a `daq_create_readoutdevice` command. Both the number of parameters and the meaning of the parameters of the command is highly device-specific. Each device which the system knows about is defined by a unique acronym such as `LC2249W` for a LeCroy 2249W ADC. A line defining such an ADC would look like

```
-> daq_create_readoutdevice ("LC2249W",1,1005,0,1,5,0)
```

meaning, in plain English, a LeCroy 2249W ADC to be read out in an event of type 1 (more about this in a moment), whose data goes into a subevent with id 1005. The ADC is to be found on CAMAC brach 0, crate 1, slot 5, and subaddress 0. This is the order in which the parameters appear. The first parameter is the device acronym, and parameters 2 and 3 tell PDAQ when to read out the device and how to store the data. This information must be coordinated with the analysis software, which needs to know those two numbers to get to the data. Those first 3 parameters (acronym, event type, and subevent id) have the same meaning for *all* devices.

The rest of the paramter list tells PDAQ how to get to the device, in this case, CAMAC branch 0, crate 1, slot 5, and subaddress 0. The meaning of the parameters and also the number of parameters is very device-specific. For example, to fully describe the VME-based G-2 readout board (which we called "Hammond" because it read out a "Hammond" board in the TEC AGS test), it takes only the VME base address of the board. Here is a Hammond board definition line:

```
-> daq_create_readoutdevice ("Hammond",1,1101,0x2000000)
```

We see that the first 3 parameters (acronym, event type, subevent id) are there (although different) like for the LeCroy ADC, but the device-specific parameters to point PDAQ to the hardware are very different.

There is another debug/test-type device in the system, which is called a `dummyDevice`. It does not read out any hardware but is generated from scratch by the processor and comes in handy to see whether or not data actually arrive at their destination. The data which the `dummyDevice` produces look like a (fictitious) ADC whose channel i holds the value i, so you can use it to test whether the channel assignments in the analysis are right. Here we use it as an example how to define a device which – except for the mandatory 3 parameters – takes no device-specific parameters at all:

```
-> daq_create_readoutdevice ("dummy",1,1020)
```

Just the event type and the subevent id are enough, there is no additional description needed to define a `dummyDevice`.

However, one commonality exists for CAMAC devices: They all have as the device-specific parameters, in CAMAC-lingo, the BCNA: CAMAC brach (B), crate (C), slot (N), and subaddress (A). For VME devices the user must look up the required parameters from a list.

4.4 Setting up a script to define devices

Rather than typing in all the definitions by hand, you will usually write a script (or a number of scripts) which will set up all the devices and other definitions. For now we assume that we have a LeCroy 2249W ADC and a LeCroy 2228 TDC in crate number 1, slots 9 and 11, respectively. We want the ADC's data to end up in subevent 1005, and the TDC in subevent 1009. A script could look like this:

```
daq_clear_readlist
daq_create_readoutdevice ("LC2249W",1,1005,0,1,9,0)
daq_create_readoutdevice ("LC2228",1,1009,0,1,11,0)
daq_list_readlist
```

What we do in the first line is to *clear* the whole readlist – we want to get rid of any definition which might be left over from our previous playing around.

Then we define the two devices, and in the end we give the `daq_list_readlist` command to see what we have done.

If you now wonder, "How do I edit this script on the VxWorks system?", you don't. The disks of your UNIX host are usually NFS-mounted on the VxWorks system, and you edit the file on the UNIX system and use it on VxWorks. Say you call this script `setup.ddf` (the `.ddf` stands for "device definition file"), you can execute the script on the VxWorks shell by typing

```
-> < setup.ddf
```

This may look strange, but it makes sense: The "`<`" is, like on UNIX, the input redirection, so we tell the shell to use that file as its input here. It will read all the lines and execute them one by one.

The VxWorks shell has some "features" which may take some getting used to. For example, the parameter for a `cd` (change directory) command must always be given in quotes, otherwise the shell will treat the parameter as a number or as a variable (and fail in this case). Now we can better understand those commands we gave to define devices:

```
-> daq_create_readoutdevice ("Hammond",1,1101,0x2000000)
```

will treat the quoted *Hammond* as a string, and the rest as plain numbers to pass to a routine called `daq_create_readoutdevice`. It is the shell which understands the `0x2000000` to be a hexadecimal number and treats it accordingly.

Now you can run the system again, this time actually reading the ADC and the TDC which we defined. We will now look how we are transferring the data to our remote UNIX system and write them to disk there.

4.5 Setting up the UNIX event server

We are leaving our VxWorks shell now, because we need to set up the *event server* on our UNIX system. The server is called `ndd_event_server`. It acts as a receiver for the data, very much like a FTP server is listening for someone to log in and transfer files.

What we have to decide now is a directory where we want the data to go, and a port number (if you don't know TCP/IP well, just think of it as the number of the channel which we want to use for our communication). The TCP/IP standard mandates that port numbers below 5000 are to be left alone, so we pick 5005 for our example here.

Before we go on, let's review the function which this server has. In the current configuration, the `ndd_event_server` receives buffers from the VxWorks system, writes them to a file, and at the same time puts the events one by one into what we call a *Data Distributor Pool* (DD). This pool is where our online monitoring processes get their data from, and any number of processes can connect to the DD pool and retrieve event data according to some strategy. One process might explicitely request that it gets *all* events in the original sequence, another process might want the data as fast as it can process them, but not necessarily all of them, another might request data from events which have been given some tag (we do not yet fully support that option). There are lots of possibilities to configure the data retrieval from the DD pool, and this business is treated at length in dedicated manual.

For now we want to concentrate on the other function of the server, the logging of the data to a file. When the server receives data from Pdaq, it writes the data to a file, which is opened when the server receives a *begin-run* instruction. This instruction is generated by Pdaq when we give the `daq_begin` command (and we told Pdaq to actually ship the data somewhere, which we haven't done so far). We are normally going to produce a large number of runs, and it would be much too inconvenient to manually give a new file name to each of them. For this reason, we use the run number to construct a file name according to a *file rule.*

This file rule is a normal string which could be used in a standard C `printf` statement, such as in this small program:

```
#include <stdio.h>
main()
{
int run = 42;
printf (run_%05d.evt\n,run);
}
```

which puts the number in a 5 character wide field with leading zero's, and yields

```
run_00042.evt
```

In the same way we give a `printf` control string as a file rule, and the server uses that string internally to construct a file name as we have just seen. If we call our `ndd_event_server` with this same rule (note the quotes to protect the string from the UNIX shell):

```
Unix> ndd_event_server 'run_%05d.evt'
```

we get the file names `run_00001.evt`, `run_00002.evt`, `run_00003.evt`, `run_00004.evt`, and so on, for runs number 1,2,3, and 4. There is no restriction what you can put into the rule as long as it can be used by `printf`; for example, you can give an explicit directory:

```
Unix> ndd_event_server '/data/ags/run_%05d.evt'
```

which will create the files `/data/ags/run_00001.evt`, `/data/ags/run_00001.evt`, and so on.

Note: It is important that the control string gives you valid filenames. The if you omit the `0` in `run_%05d.evt` and write `run_%5d.evt`, you will end up with file names which have spaces in the middle, like `run_    1.evt`, which are really ugly to work with. Also, just giving `run_%d.evt` will result in filenames like `run_1.evt`, and when you go above 10, it will become `run_10.evt`, and a directory listing will not give them in sequence. Try to have a `%05d` somewhere in the rule.

The server has more parameters, which are, in this order, the port number and the name for the DD pool. If you omit all parameters, they default to

```
Unix> ndd_event_server 'run_%05d.evt' 5001 'test'
```

The last parameter is usually left as the default; in a more sophisticated setting it is possible to have more than one DD pool on more than one node.

Let's see the server at work now. We start it on port 5005:

```
Unix> ndd_event_server 'run_%05d.evt' 5005 &
```

The server is now ready to accept a connection.

Now we have to go back to our VxWorks terminal and tell Pdaq where and on which port it will find its server. The parameters are given by the `daq_set_host` command. Since VxWorks does not normally have access to a name server, we will have to give the IP address of our UNIX machine directly; here we work on `phoncs1.phy.bnl.gov`, which is `130.199.22.90`:

```
-> daq_set_host ("130.199.22.90",5005)
```

Again, the IP address is given as a string, and the port is given as a number.

Now we can establish a connection with the server by giving the `daq_open` command. This command does not yet open any files, it will just have Pdaq make contact with the server and tell it to stand by.

```
-> daq_open
ip_adr 130.199.22.90
opening protocol...
opening Stream...
opening socket for 130.199.22.90 on port 5005
Stream opened, status= 0
value = 0 = 0x0
->
```

On the UNIX side, since we did not redirect the server's output to a file, we see a message on the screen:

```
new connection accepted from iocondev5.phy.bnl.gov
```

We can see what we have done at any time with the `Status` command, which gives a lot of information meant for the expert:

```
-> Status
```

```
Run not active
Event number 0
No volume limit
No event number limit
Protocol is open
Buffersizes are 131072 and 131072 Bytes
value = 0 = 0x0
->
```

The important message here is that the "Protocol is open", meaning that a connection has been established.
Now it's time to begin a run:

```
-> daq_begin 1
```

and on the UNIX side, we see the message

```
opened file run_00001.evt
```

just the way we wanted it. If there are any pulses now arriving at the trigger input, we are getting events right now (in this example we don't get any triggers).
Now we end the run:

```
-> daq_end
```

and on the UNIX window we see

```
 end run -- number of events 2
```

We can now go and begin run number 2, or another number, as we like it. At the end of the day, when we think we are done, we can break the connection with the server again:

```
-> daq_close
```

And on the UNIX side we see the message

```
Socket closed, number of buffers: 1
```

So for this example we didn't transfer a lot of data, but it worked. If you wonder what those two events have been, they are the begin-run and the end-run event of run number 1 (each run has those two events, even if no data events have been triggered).

There is no need to close the connection; once established, you can begin and end as many runs as you wish. It is just good practice to close the connection if you are planning for a extended break, or are planning to reboot one or both machines.

4.6 Putting it all in a script

Let's wrap this little procedure up in a few scripts so that we can re-do everything without typing too much. Note that each line starting with a number sign (#) is treated as a comment by the VxWorks shell, so we can add a few comments so that we remember what we have been doing. We extend the script from above to read

```
# -- Pdaq example setup script for the manual --
#
# initialise the system
daq_init
# set the host IP address (phoncs1.phy.bnl.gov) and port 5005
daq_set_host ("130.199.22.90",5005)
#
# clear the readlist
daq_clear_readlist
# create an ADC in crate 1, slot 9
daq_create_readoutdevice ("LC2249W",1,1005,0,1,9,0)
# and a TDC in crate 1, slot 11
daq_create_readoutdevice ("LC2228",1,1009,0,1,11,0)
# and list what we have done
daq_list_readlist
# now open the connection:
daq_open
# done -- we are now ready to begin a run.
```

Now this looks quite nice, however, one would want to break it up into more scripts which do only a thing at a time. For example, make a file called `devices.ddf`:

```
# set up the read list for our example
# clear the readlist
daq_clear_readlist
# create an ADC in crate 1, slot 9
daq_create_readoutdevice ("LC2249W",1,1005,0,1,9,0)
# and a TDC in crate 1, slot 11
daq_create_readoutdevice ("LC2228",1,1009,0,1,11,0)
# and list what we have done
daq_list_readlist
```

and a file `setup.ddf`:

```
# -- Pdaq example setup script for the manual --
#
# initialise the system
daq_init
# set the host IP address (phoncs1.phy.bnl.gov) and port 5005
daq_set_host ("130.199.22.90",5005)
# now execute the devices.ddf file to define the read list:
< devices.ddf
# now open the connection:
daq_open
# done -- we are now ready to begin a run.
```

As we see, we can easily nest the scripts. We can execute `setup.ddf` at the very beginning, perhaps even set it up in a way that it is executed at boot time. Later we can change the read list and execute `devices.ddf` without going through the whole initialisation procedure again.

5 Advanced configurations

5.1 Different Event types

In one of the previous examples we briefly mentioned a begin-run and an end-run event, which are generated automatically by the Pdaq system. The concept of different event types comes in when you want to read out different things at different moments. For example, you normally want to read out your ADC's and other devices which you have connected to your detector at the AGS beam line. At the end of the AGS spill, however, you want to know what the beam intensity was in the last spill, so you count your raw beam triggers (the ones which were generated by the trigger logic, not the ones accepted by Pdaq) in a scaler and read this out in a different event type, a *spill-off event*. You make sure that each time you receive a signal from the AGS that the spill ended, you trigger such an event, and it will go and process a read list which is totally different from the readlist for a normal data event.

It's easy to confuse those event types with the distinction between, say, "physics"-type events and "calibration"-type events, which is not the case: *Different event types have different read lists.*

So in our example with the beam counter the readlist for a type 1 ("data") event is the ADC's and other stuff to read out your detector, while for a "spill-off" event it is the scaler, which is a completely different read list.

That said, we have to admit that nobody has made use of a spill-off event so far; the only time Pdaq has been used at the AGS, it was decided that the setup can live without that. But here is a `ddf` example which would define a setup like the one above. It uses the `LC2551` device (readout the LeCroy 2551 scaler) and the `LC2551CLR` device (don't read out, but reset the scaler):

```
# set up the read list with a spill-off event
# clear the readlist
daq_clear_readlist
# create an ADC in crate 1, slot 9
daq_create_readoutdevice ("LC2249W",1,1005,0,1,9,0)
# and a TDC in crate 1, slot 11
daq_create_readoutdevice ("LC2228",1,1009,0,1,11,0)
# now define a LeCroy 2551 scaler for the spill-off event (type 16):
daq_create_readoutdevice ("LC2551",16,1001,0,1,15,0)
daq_create_readoutdevice ("LC2551CLR",16,0,0,1,15,0)
# and list what we have done
daq_list_readlist
```

So we see that we define the LeCroy 2551 scaler to be first read out (and then reset) in a type 16 event, which is the type of a spill-off event (since the `LC2551CLR` device just resets the scaler but doesn't generate any data, the subevent id is ignored and set to 0 here). It would be our job now to make sure (with additional trigger logic) that such an event is triggered at the end of the AGS spill.

Two comments at this point. First, you cannot invent new event types on your own; you can only use those which are already defined in Pdaq, and you should not abuse those events for something else (because Pdaq assumes that those events are used for what they are intended). The reasons for that are that those event types have to be coordinated with the analysis part of the software, and, more importantly, Pdaq takes additional action when some of the events are triggered. For example, if you trigger a Spill-off event, Pdaq will perform a buffer switch for its internal output buffers. In this way we synchronize the buffer-switching with the spill structure, so we don't waste time for buffer switching overhead in the middle of a spill.

The other comment is about subevent id's. In principle the id's need to be unique only within a given event type, so you could define

```
daq_create_readoutdevice ("LC2228",1,1009,0,1,11,0)
# now define a LeCroy 2551 scaler for the spill-off event (type 16):
daq_create_readoutdevice ("LC2551",16,1009,0,1,15,0)
```

using the 1009 id for both subevents. This works, because you will have either a data (type 1) or a spill-off (type 16) event, and no subevents with the same id within the same event. However, since there are so many numbers to choose from, we should avoid the confusion that an id can have different meanings in our analysis software and just pick unique numbers across event types.

Now let's come to an actual example from the TEC beam test, where different event types have been used, in this case the begin-run event. This event is just a normal event which can have a read list, except that it is "triggered" automatically when we type `daq_begin`. So it would be ok to define an ADC to be read out (although it would not do us any good, since there was no hardware trigger and no gate generated). The real purpose of this event is to initialize devices at the begin of a run and to read out and record device settings when the run starts.

Let's assume that you have a software-configurable CFD (which we don't have in Phenix); the GSI-built CF8001 comes to mind here. Of course this is not normally a readout device; it is usually an important part of your trigger logic, and since the thresholds and other parameters define how your trigger works, it would be good to know what those parameters were (assuming they are set by some other piece of software). We could now write those parameters down in a log book, or put them into a database, but there is a certain risk that the log book gets lost, or the database is corrupt, or something else goes wrong, so it might happen that the information what those parameters were is not available.

This is where our begin-run event comes in: We provide software to read out the CFD parameters like an ADC or another device, that is, turn the CFD into a readout device, and read it in the begin-run event:

```
daq_create_readoutdevice ("CF8001",9,1099,0,1,18,0)
```

The begin-run event has the type 9, so what we do here is to dump all the CFD parameters into subevent 1099 of the begin-run event. Normally this information just sits there and is usually left alone, but if something turns out to be fishy in the analysis later, we can actually go and see if there has been a change in the parameters which has not been logged properly.

Now for the TEC setup at the AGS (figure 3) we had such a device which made use of the begin-run event. We read out the "Hammond board" through 2 G-2 boards in VME, and for normal data taking we would just read the G-2's and some CAMAC stuff. However, in order to be initialized properly, the Hammond board needs to be loaded with a few (5 or so) values to define some voltages, thresholds and such. We could have done it with a standalone program, but inventing a new device for our readlist was much more elegant:

```
daq_create_readoutdevice("Hammondsetup", 9,1999,0x2000000)
```

When we attempt to read out this `Hammondsetup` device, it first goes through the G-2 board to download a few (standard) parameters to the Hammond board. Then it dumps a copy of the data to subevent 1999 (in the begin-run event), so we not only initialize the board, but we also have a log what we loaded there. So each time we start a new run, we re-initialise the board with the proper parameters, and log the values in the begin-run event.

The command to create the `Hammondsetup` device also introduces a new feature. We can give the command as shown above, and we will create

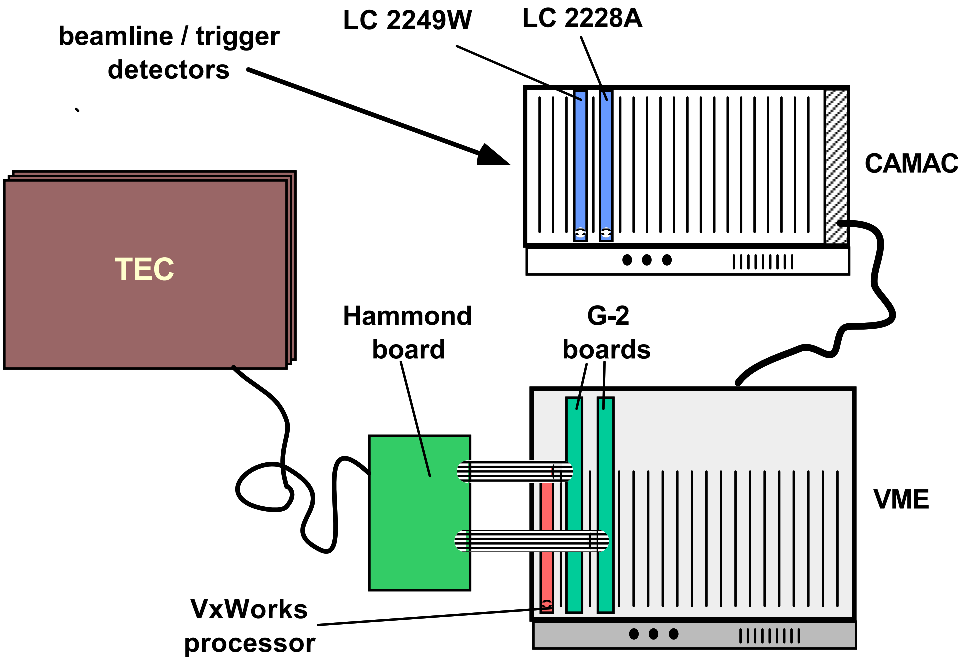

Figure 3: The setup for the TEC test beam at the AGS

a device object which will load (and log) a standard set of parameters. However, we can give it more parameters, and can create a `Hammondsetup` device which loads custom parameters to the board:

```
daq_create_readoutdevice("Hammondsetup", 9,1999,0x2000000,
0x5A22,0x4BDB,0x088C,0x2998,0x0000)
```

(We had to break this command across lines, but it is meant to be a single line only.) So we have a way to play with the settings of this board, which was one of the purposes of that AGS beamtime. Here is a copy of the whole setup file which was used:

```
daq_clear_readlist()
#CAMAC slot 3
daq_create_readoutdevice("LC2249W", 1,1500,0,1,3,0 )
#CAMAC slot 11
#daq_create_readoutdevice("LC2228", 1,1501,0,1,11,0 )
```

```
  # register the Hammond board's setup values.
daq_create_readoutdevice("Hammondsetup", 9,1999,0x2000000,
0x5A22,0x4BDB,0x088C,0x2998,0x0000)

  # register the two G-2 boards.
daq_create_readoutdevice("Hammond", 1,1101,0x2000000)
daq_create_readoutdevice("Hammond", 1,1102,0x2000020)
```

We can see the definitions for the two CAMAC devices and two Hammond boards for event type 1, and a `Hammondsetup` for event type 9.

5.2 Reading a device in more than one event type

You might think that you can read out a device in one event type only. This is not the case (although that's what you normally do). There are cases when you might want to read devices (in particular scalers) in more than one event type, and this section is meant to show you that it is quite alright to do so.

Let's go back to our scaler example from the previous chapter. There we read out (and reset) a LeCroy 2551 scaler with each spill-off event:

```
# ... stuff deleted
# now define a LeCroy 2551 scaler for the spill-off event (type 16):
daq_create_readoutdevice ("LC2551",16,1001,0,1,15,0)
daq_create_readoutdevice ("LC2551CLR",16,0,0,1,15,0)
```

If you need the count very precisely (for example, if you need that number to calculate an absolute cross section), you should make sure that you reset the scaler at the begin of the run, and also read it at the end of the run to account for the counts between the last spill-off event and the actual end of the run:

```
# ... stuff deleted
# clear the scaler at the begin-run event (type 9):
daq_create_readoutdevice ("LC2551CLR",9,0,0,1,15,0)
# now define a LeCroy 2551 scaler for the spill-off event (type 16):
daq_create_readoutdevice ("LC2551",16,1001,0,1,15,0)
daq_create_readoutdevice ("LC2551CLR",16,0,0,1,15,0)
```

```
# and for the end-run event (type 12) as well:
daq_create_readoutdevice ("LC2551",12,1001,0,1,15,0)
```

So we reset (but don't bother to read out) the scaler at the begin of the run, read and then clear it at each spill-off event, and read it out once more at the end of the run. We use the same subid, since the subevent id has the same meaning in the different subevents.

5.3 Using raw CAMAC functions

Occasionally you want to trigger a specific CAMAC function when a certain event is triggered. We implemented a `CAMACFUNCTION` device which does not generate any data or an actual subevent in the data stream but executes a single CAMAC function when PDAQ attempts to read it. The syntax is

```
daq_create_readoutdevice ("CAMACFUNCTION", event type, subevt id, B,C,N,A,F,Data)
```

Since this "device" does not generate any data, the subevent id is ignored (we have it only to preserve the standard parameter list for CAMAC devices). In order to generate a pulse on output 1 of an output register at the begin-run event, you can define (we assume it's in slot 17):

```
daq_create_readoutdevice ("CAMACFUNCTION", 9, 0, 0,1,17,0,16,1)
daq_create_readoutdevice ("CAMACFUNCTION", 9, 0, 0,1,17,0,16,0)
```

So we write (F=16) a "1" to the register, setting output number 1 high, and then set it to 0 again, setting it low, so we generate a pulse.

Note: You should not use the same output register which is used for the busy logic. If you do not know exactly what you are doing, you may cause severe side effects.

You deal here with raw single CAMAC fuctions, so you have to look up the proper function codes and data it takes to achieve the desired effect.

Let's assume that we have counters (displaying numbers for you to see, not counters to be read out) counting your raw beam triggers in the given AGS spill, the events accepted in the spill, and the events accepted in the whole run (which is nice to have to really see what you get). You will want to reset the events-per-run counter each time you begin a run, and you want to reset the per-spill counters at the begin of each spill. You can generate the reset pulses from the output register with the following definitions:

```
# output 1 resets the counter at the begin of each run
daq_create_readoutdevice ("CAMACFUNCTION", 9, 0, 0,1,17,0,16,1)
daq_create_readoutdevice ("CAMACFUNCTION", 9, 0, 0,1,17,0,16,0)
# output 2 resets the counter at the begin of each spill
daq_create_readoutdevice ("CAMACFUNCTION", 8, 0, 0,1,17,0,16,2)
daq_create_readoutdevice ("CAMACFUNCTION", 8, 0, 0,1,17,0,16,0)
# output 3 resets the counter at the begin of each spill
daq_create_readoutdevice ("CAMACFUNCTION", 8, 0, 0,1,17,0,16,4)
daq_create_readoutdevice ("CAMACFUNCTION", 8, 0, 0,1,17,0,16,0)
```

We could use output 2 to reset both the per-pill counters, but in this way you have the possibility to redefine in software when the counter is reset (and you save a fan-in/fan-out module).

5.4 Setting event number or data volume limits

If you begin a run, it will normally remain active until you type `daq_end`. If you go for a coffee and you forget to end the run, you may end up with a data file on the Unix system which is several GB big, and this can be hard to work with. For this reason we support an automatic end of the run when certain criteria are met. This can either be the number of events in a run, or the data volume produced in a run, or both. So you can run in a mode where the run is ended after, say, 5000 events, or when the data volume exceeds 300MB. If you look at the output of the `Status` command on the VxWorks shell, which we showed in one of the previous chapters, it said

```
-> Status
Run not active
Event number 0
No volume limit
No event number limit
Protocol is open
Buffersizes are 131072 and 131072 Bytes
value = 0 = 0x0
->
```

So by default we do not have any limit set. We can set it by the commands

```
daq_set_max_events 5000
daq_set_max_volume 300
```

The unit for `daq_set_max_volume` is Megabytes, and now the `Status` command tells us

```
-> Status
Run not active
Event number 0
Will end run at 300 MB -- size is 0 Kb
Will end Run after 5000 events
Protocol is open
Buffersizes are 131072 and 131072 Bytes
value = 0 = 0x0
->
```

We can get rid of a limit by setting it to 0:

```
daq_set_max_events 0
daq_set_max_volume 0
```

It is good practice to set a volume limit for your data files; you will end up with files of approximately the same size if you have a continuously running experiment.

Note: The system does not provide a automatic re-start of a run after it was ended automatically. We will show in chapter 6 how to achieve this.

5.5 Adjusting buffer sizes

The output of the `Status` command on the VxWorks shell said

```
-> Status
Run not active
Event number 0
No volume limit
No event number limit
Protocol is open
```

```
Buffersizes are 131072 and 131072 Bytes
value = 0 = 0x0
->
```

So what are those buffer sizes, which "are 131072 and 131072 Bytes"? Pdaq does not transfer the data to the UNIX host event by event, but collects them in buffers, which are then transferred in one big chunk (this reduces a lot of overhead with setting up the transfer). The buffers are always a multiple of 8192 (8k) bytes ($131072 = 16 \times 8192$). Normally there is no reason to change those sizes, but there are situations where you want to do this. Let's say you run a setup with a very slow trigger rate and/or a very small event size, for example, a cosmic-ray measurement. Then it takes a long time for a buffer to be filled and to be transferred. Your online monitoring process will wait for a long time for events to appear, and then there is a burst of many events in a very short time, and then again nothing for a long time. So you can change the size of the buffers and with it the time it takes to fill and then transfer them and get a steadier data flow to your UNIX system.

The sizes must be a multiple of 8192, but the system does not let you adjust any other value; if it is not a multiple of 8192, it will be rounded up. This leads to the most common way of adjusting the size:

```
-> daq_set_maxbuffersize 1
max buffersize is 8192 Bytes
```

This will set the buffer size to the smallest possible value (the 1 is rounded up to 8192). 0 is a special value and means "set the default size", so we go back to the original value:

```
-> daq_set_maxbuffersize 0
max buffersize is 131072 Bytes
```

Also, this default value represents currently the highest possible value which you can adjust; giving a number larger than that gives the default value again. The values are set the same for both output buffers; the system does not allow you to set them differently.

5.6 Faking triggers

Occasionally you want to start your readout chain without setting up a full hardware trigger. We used this possibility in the magnet-mapping setup, where we read only some CAMAC-based digital voltmeters, which do not require a gate or stop signal and can be read at any time. The command

```
-> daq_fake_trigger 5
```

will "fake" 5 triggers with a rate of about 1Hz by internally calling the routine which would be called by the interrupt handler if we had received a real interrupt. This makes no difference for the VxWorks processor; it does not know that it has been woken up "manually". The processor goes through the whole routine of trying to clear the busy latch through a pulse on the output register, so we still need a CAMAC crate for such a setup. In chapter 6 we show an application of this command in more detail.

Keep in mind that this command bypasses the whole hardware trigger logic and works in software only. If you have ADC's or TDC's which need a real gate or stop signal, they won't get any, because there was no real pulse in this case. If you need a real test pulse, read on.

5.7 Getting a real NIM pulse

If you want to test your setup, you normally hook up some NIM pulser (such as a LeCroy InstaPulser) to the start input. However, you may not always have such a pulser handy, or you may want *exactly one* (or a certain mumber of) real hardware pulses. For your convenience, we implemented the `pulser` command, which gives you NIM pulses at output 1 of the output register (maybe you wondered why the reset-busy pulse comes from output 2). This command is fully independent from the Pdaq system. It does not do anything behind your back, it just gives you a NIM pulse which you can use for any purpose, for example, to trigger your data acquisition. The following command gives you exactly 100 pulses (at a fixed rate of about 1 Hz):

```
-> pulser 100
```

5.8 Controlling the event server

On the UNIX side, we learned how to start the **ndd_event_server** which receives the event data from Pdaq. We will introduce you to two commands which talk to the server after it has been started. Let's assume we start it, as before,

```
Unix> ndd_event_server 'run_%05d.evt' 5005 &
```

We can use the **ndd_display** command to see what the parameters are:

```
> ndd_display
status = 0
Logging is enabled
current filename:
filename rule:    run_%05d.evt
>
```

If we had a file open (a run active), we would also see the current filename. Say we want to change the file rule and store the data in the directory **/data/ags/run_%05d.evt** instead. We can reconfigure the server with the **ndd_setfilerule** command:

```
> ndd_setfilerule '/data/ags/run_%05d.evt'
> ndd_display
status = 0
Logging is enabled
current filename:
filename rule:    /data/ags/run_%05d.evt
>
```

The change takes effect when we open the *next* file, so it does not effect a file which is currently open.

6 The Unix shell mode

The Unix shell mode has been implemented in order to be able to control the whole system from one single process. Apart from the convenience, this enables you to steer the data acquisition based on what you find in your online monitoring in an automated fashion. In the example which we already mentioned above, this mode was used to automatically adjust a peak in a spectrum to be at a certain mean value, which means beginning a run, look at the peak value, adjust a high voltage, re-run the DAQ, look again, and go through several iterations if necessary. Without the possibility to let the Unix process control the DAQ, this would require a lot of tedious manual typing.

This Unix shell mode is one step short of using the PHLOPS-mode CORBA communications mechanism; it uses the better-known RPC (Remote Procedure Call) mechanism with the advantage that this is available on virtually each Unix system.

With very few exceptions, the UNIX commands have the same name and take the same parameters as their VxWorks equivalents. Due to the different parameter passing, each command on Unix just takes space-delimited parameters. The VxWorks command

```
-> daq_create_readoutdevice ("LC2249W",1,1005,0,1,9,0)
```

becomes

```
Unix> daq_create_readoutdevice LC2249W 1 1005 0 1 9 0
```

so this is very similar. There are some exceptions, mostly due to the way the shell treats numbers, in particular with the hexadecimal numbers which we saw in the definition of the Hammond board. Because the VxWorks shell treats numbers as such, and understands hexadecimal numbers, we were able to write

```
-> daq_create_readoutdevice ("Hammond",1,1101,0x2000000)
```

and the `0x2000000` is treated properly. The Unix utilities do not do this automatically, so you will need to write the decimal value

```
Unix> daq_create_readoutdevice Hammond 1 1101 33554432
```

The Hammond is, so far, the *only* device with this problem, and probably never used again, so it was decided not to fix this.

Before you try it and start to type Unix commands, there is one additional definition to make in order to tell the Unix system where it will find the VxWorks processor. Our Unix system has ready access to a name server, so we can define

```
setenv DAQ_RPC_HOST iocondev5.phy.bnl.gov
```

We are all set now.

We should mention that the `daq_list_readlist` command is not available from Unix (that is because the devices send their identification string to VxWorks standard output, and changing that would be very involved). There are two more commands available here which are not available from the VxWorks shell, `daq_info` and `daq_wait_end`.

`daq_info` gives you a brief one-line summary like the VxWorks `Status` gives.

`daq_wait_end` makes it possible for the Unix shell to wait for the end of a run. It is meant for use in automated scripts, which we will look at now.

6.1 Automated Unix shell scripts

With the commands available to the UNIX shell, we can make scripts to make routine tasks easier. As an example, we show a script `measure.com` which was used for the magnet mapping setup.

```
#!/bin/sh
daq_begin $1
sleep 1
daq_fake_trigger 5
sleep 1
daq_end
daq_wait_end
```

As we explained before, the magnet-mapping setup contained only "passive" readout devices. We used 3 SAM5305 modules, which are 32-channel digital voltmeters in CAMAC and which can be read at any time – they do not need a gate or anything and so we could run without any external hardware trigger logic and fake the triggers.

So we would run the script

```
measure.com 1223
```

and it would open run 1223, take 5 fake triggers, and end the run again. The reason why we wait for the end of the run *after* the `daq_end` command is that it may take a few seconds after the command until the run has truly ended (we still need to transfer the buffer which has been filled, and so on), so it is good practice to wait and make sure that the run has really ended. If the run has already ended, the `daq_wait_end` command returns immediately.

6.2 Automatic start of the run

When we discussed the data volume or event number limits in chapter 5.4, we said that Pdaq does not provide a mechanism to start another run after the previous run ended automatically due to a volume or event number limit. We can add this functionality with a simple Unix shell script (called `run_control` here) which starts a run, waits for the automatic end, increments the run number, and starts the next run:

```
#!/bin/sh
#

#  set the volume limit to 300MB
daq_set_max_volume 300
#  start with the run we gave as a parameter
run=$1
while :
do
     echo "starting run  $run"
     daq_begin "$run"
     daq_wait_end
     echo "run  $run ended"
     run=`expr $run + 1`
done
```

So now you type

```
> run_control 100

starting run  100
run  100 ended
starting run  101
run  101 ended
starting run  102
run  102 ended
```

and sit back and enjoy the data taking.

7 Reference Section

7.1 Currently supported event types

The current version of PDAQ is capable to generate the following event types:

1 data event (which you use to read out your electronics)

8 spill-on event (which you can generate at the begin of the spill)

9 begin-run event (generated automatically at the begin of a run)

12 end-run event (generated automatically at the end of a run)

16 spill-off event (which you can generate at the end of the spill)

Each of the event types has its own independent read list.

7.2 Currently supported device types

The list of currently supported readout devices is expanding rapidly; new devices can easily be implemented. We keep the up-to-date list on the Web under the ONCS pages, so if a device is not in the list, check the Web pages first.

The devices in the following list are given with a brief description of the parameters. All device definitons take the acronym, event type, and Subevent id as their first 3 parameters, and then device-specific parameters to tell

Pdaq how to find the device. "CAMAC parameters" refers to the standard paramater set which is the same for *all* CAMAC devices: Branch, Crate, Station, Subaddress. For VME-based devices there is no standard set of parameters.

AD811 The Ortec peak-sensing ADC. CAMAC parameters.

SAM5305 The BIRA "Smart Analog Monitor", a 32-channel digital voltmeter in a CAMAC module. CAMAC parameters.

LC2249W LeCroy 2249 12-bit (wide gate) charge-sensitive ADC. CAMAC parameters.

LC2228 LeCroy 2228 TDC. CAMAC parameters.

LC2251 LeCroy 2251 scaler readout. CAMAC parameters.

LC2251CLR LeCroy 2251 scaler reset. The subevent id is ignored because this "device" doesn't produce any data. CAMAC parameters.

CAMACFUNCTION Execute a raw CAMAC function. The subevent id is ignored because this "device" doesn't produce any data. CAMAC parameters.

mizar The WA98 MIZAR DSP board, which is used to read out the lead glass front-end boards. Device-specific parameter is the board id (0-7).

```
-> daq_create_readoutdevice("mizar",1,1130,6)
```

dl600 The DL600 VME board built by the University of Heidelberg. It is used to read out the angle endcoder in the magnet-mapping setup. Device-specific parameter: The base address of the board.

```
-> daq_create_readoutdevice("dl600",1,1150,0x55)
```

Hammond The Hammond board readout through one or two G-2 VME boards. For a complete readout 2 G-2 boards are needed. Device-specific parameter: The base address of the board (which is adjusted through jumpers on the board).

```
-> daq_create_readoutdevice ("Hammond",1,1101,0x2000000)
```

Hammondsetup This device configures the Hammond board through the serial output of a G-2 board, and stores the loaded set in the subevent. The configuration consists of 5 numbers. Device-specific parameters: The base address of the board. Optional: A set of the configuration data. If the data are omitted, a standard set is loaded.

```
-> daq_create_readoutdevice("Hammondsetup", 9,1999,0x2000000)
```

or, with custom parameters

```
-> daq_create_readoutdevice("Hammondsetup", 9,1999,0x2000000,
0x5A22,0x4BDB,0x088C,0x2998,0x0000)
```

dummy A "dummy" device which is generated by the processor and is useful to test the data flow. It looks like an ADC whose channel i holds the value i, so you can check the channel assignments in your analysis software. No device-specific parameters.