P.Gorbunov 25 May 2004
Last update: 31 May 2004

Description of BeamCrate_t and BeamData_t classes

The class BeamCrate_t is intended for controlling the H6 beam DAQ from a remote C++ application. In includes methods to start, stop, pause and resume a run, set the parameters for the run to be started (configure a run), get the DAQ status and, optionally start the beam DAQ process(es). The class BeamData_t is indended for a separate recorder application which only receives the beam ROD fragment data stream and does not deal with the run control. All details of communication and synchronization with the beam DAQ are incapsulated in the class objects. The implementation of the classes is based on the ControlHost package [1].

1. Common features

1.1 `#include "BeamDaq.h"`

The header file contains class declarations and defines constants and auxiliary types associated with the classes.

1.2 Declarations

BeamCrate_t and BeamData_t objects are created by declaring them with one of available kinds of declarations (constructors). No connection[1] is attempted by the constructors. For the data object, a permanent connection is established by the **`subscribe`** method (Section 3.2.1), while the control object spends most of the time unconnected. The run action methods connect to the host each time they are invoked and disconnect on a completion of the action.

1.3 Restrictions

- The classes BeamCrate_t and BeamData_t are mutually exclusive: a single application cannot be a controller and a receiver simultaneously. This restriction is not fundamental and can be removed in future releases, if needed.
- Only one object (control or data) per application can be declared. This restriction is more serious and is related to the ControlHost [1] implementation I am currently using. It allows only one connection to the ControlHost server (*dispatcher*) per process, therefore any new subscription to the dispatcher overrides the existing one.

1.4 Destructors

When the objects are destoyed, the existing connections to the beam host are terminated.

1 The term “connection” means a subscription to the ControlHost server (*dispatcher*) normally running on the DAQ machine. See [1] for further details.

1.5 Return codes

Most of BeamCrate_t and BeamData_t methods return an integer error code (zero value always means a success). The return code values are defined in the header file BeamDaq.h as follows:

```
OK=0,                   // success
err_CONN=1,             // disconnected or a failure to connect
err_DAQ=2,              // DAQ is not connected
err_RUNPAR=3,           // error in run parameters at RunStart
err_TIMEOUT=4,          // time-out
err_ILLEGAL=5,          // illegal command (for the context)
err_CONTEXT=5,          // the same
err_GET=6               // error in getting data or an illegal subscription
                        // (should never happen!)
run_STOP=7              // BeamData_t:: get_rod ("run stopped" interrupt)
```

1.6 Beam DAQ and run states

The BeamCrate_t and BeamData_t methods refer to the beam DAQ or run *states* or *contexts.* The following states (contexts) are defined:

"DaqQuit"	DAQ is about to stop after receiving a command to quit the DAQ; the DAQ issues the "DaqStopped" message, then disconnects from the dispatcher and actually stops.
"Aborted"	DAQ is aborted by a kill command or Ctrl-C and is about to stop.
"Stopped"	no run is ongoing: the state between two runs; the beam DAQ starts in a "Stopped" state
"Starting"	a transitional state after receiving a command to start a run, no triggers are sent yet
"Running"	normal running state: physics triggers are sent to FEBs
"Paused"	a state after receiving a command to pause a run: the triggers are temporarily blocked
"Stopping"	a transitional state after receiving a command to stop a run: the triggers may be still coming, but will be stopped as soon as possible.

The corresponding public `static const string` members `s_DaqQuit, s_Aborted, s_Stopped, s_Starting, s_Running, s_Paused` and `s_Stopping`, as well as `s_Unknown="unknown"` are available from BeamDaq.h.

1.7 Common methods

A few methods are common to both classes (Fig. 1). They can be used to control the debug print level, inquire the status of the connection and the beam DAQ and drop the existing connection.

`void debug(const int& level)`

Action: turns on/off the debug putput to stdout
Input parameter:

`level`	0=print off; 1=print on; 2=some extra debug print (e.g., from BeamData_t::get_rod)

Return value: none

`void disconnect(void)`

Action: disconnects the object from the dispatcher and, thereby, from the beam DAQ.

Input parameter: none
Return value: none

`int check(const int& print)`

Action: contacts the dispatcher and checks that the beam DAQ is also connected.

Input parameter:

`print`	0 – quiet, no output to stdout; 1– verbouse, reports the details to stdout

Return value:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected.

`int RunStatus(string& status)`

Action: gets the current run status from the beam DAQ. A behaviour of this method is somewhat different for BeamCrate_t and BeamData_t classes. For a BeamData_t object, it normally causes no extra communication with DAQ and simply tells the last run status received from the beam DAQ while waiting for data. On the contrary, a BeamCrate_t object (unconnected in its normal state) contacts the DAQ and gets a fresh current status.

Input parameter: a reference to a variable which will hold the run context value.

Return values:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected; `err_TIMEOUT` – time-out.

`context`	One of “Stopped”, “Starting”, “Running”, “Paused”, “Stopping”, “DaqStopped” or “uncknown” (in case on errors).

2 BeamCrate_t class (run control)

The class is intended for remote run control applications. Most of the time, a BeamCrate_t object is disconnected from the dispatcher and connects only when the controling application decides to execute a run control or query action. This approach avoids an accumulation of obsolete run_status reports from the DAQ and does not cause a significant overhead. The available public methods are listed in Fig. 2.

2.2 Run parameters

The type `RunPar_t` defined in `BeamDaq.h` is used to transfer a full set of run parameters to the beam DAQ . Here is the definition:

```
typedef struct {
           int number;        // run number
           int type;          // run type (0=phys; 1=calib; 2=special
           int subtype;       // sub-type, any value
           int data_store;    // 0=none; 1=/raid; 2=dispatcher (@pcfcal02)
           int beam_mom;      // beam momentum in GeV (can be <0)
           int beam_par;      // e.g., 1=e+,2=pi+,3=p+,4=mu+,
                              // 11=e-,12=pi-,14=mu-
                              // -ve=unknown/irrelevant
           int beam_spot;     // beam spot ID (can be <0)
           int max_events;    //
           int max_bursts;    //
} RunPar_t;
```

The values for the beam particle type and the beam spot ID are not used directly in the beam DAQ (they are only copied to the run header). The other parameters are important. The combination of the run type and run subtype is transformed by the beam DAQ into the name of the configuration file[2] containing all technical settings for that kind of runs – like the FEB and calibration board settings, CAMAC configuration and trigger options. The last two parameters `max_events` and `max_bursts` defining the default run stop condition may both have non-zero values, but at least one of them should be positive. Unless the run is stopped explicitly by the parent application, beam DAQ would stop the run itself and report the corresponding condition to a receiver (see Section for further details).

2.3 Constructors

A BeamCrate_t object is created by declaring it with any of these three kinds of declaration:

```
  BeamCrate_t(void);
  BeamCrate_t(const string& host);
  BeamCrate_t(const string& host, RunPar_t run_parameters);
```

where `host` is the name of the beam DAQ machine (currently, pcfcal02.cern.ch). In the declaration without arguments, the host name is taken from the environment variable BEAM_DAQ_HOST. The third kind of declaration permits to load the initial set of run parameters, while the first two declarations preset the run parameters with the default (meaningless) values:

```
           number = -9999
           type   = -1
           subtype= -1
           data_store = 0
           beam_mom = -9999
           beam_par = -1
           beam_spot = -1
           max_events = -1
           max_bursts -1
```

2 Currently, `/run/daq/RunTypes/.`*`(type+1)`*`/.`*`subtype`*`.par`, for example `/run/daq/RunTypes/.2/.79.par` for the run type 1 and subtype 79.

2.3 Latency of BeamCrate_t methods

An important feature of all run control and run quiry methods is that they have a considerable latency due to the fact that the beam DAQ is running in a “Please, don't disturb” mode. This means that it spontaneously reports its status only at a limited number of *check-points* during the SPS cycle: at SoS and EoS (start and end of spill) and between the spills with ~1Hz frequency. The external commands are also interpreted only at these check-points, unless the run state is *“Running”* (when *“Running”*, the commands are interpreted only at SoS and EoS). The commands arriving between the “check-points” are buffered. If, for example, the “RunPause” command arrives just after the EoS, then it will take up to 12 s for a run to be paused. A response to “RunResume” command is faster: the maximun delay is ~5 s (if the command was issued during a spill). The “RunStart” command has an additional kind of latency. After accepting it, the beam DAQ goes to the *“Starting”* state which can last up to ~2 SPS cycles, until all necessary run-start configuration and calibration procedures have been completed. The LArg triggers are unblocked only when the run switches to the *“Running”* state.

For a “RunStop” and “RunPause”, the latency implies that TDAQ should be prepared to receive a bunch of event fragments (both from Larg ROD and Beam) even *after* issuing these commands. The data streams from the RODs are garanteed to stop only after the beam DAQ has reported *“Stopped”* or *“Paused”* states, respectively (see Sections 2.5 and 2.9).

An additional latency can be caused by the run context checking in the **`DaqQuit, RunStop`** and **`RunPause`** methods (Sections 2.4-2.5) if they are invoked with the parameter “wait” . The “nowait” invokation is, on average, faster but less safe. First, it might require a subsequent **`WaitFor`** method with an explicit specification of the expected context. Second, if the context turns out to be wrong for the command, then **`WaitFor`** will return a time-out condition, after a significant delay. Therefore, the “nowait” versions should be used only if the correctness of the run context for Quit, Stop and Pause commands is garanteed by the application's logic. The latency of the Resume command is practically insensitive to the context checking.

2.4 Methods

2.4.1 ***`void BeamCrate_t::DaqRun(void)`***

Action: attempts to start the beam DAQ (and the dispatcher, if needed) remotely. The code issues the shell command **`daqRun nolog`** which does the job. The script with that name must be accessible via the PATH environment variable. In order to get everything running (for example, on ATLROS-H6) the following declarations have to be added to .bashrc (or .zshrc):

```
PATH=$PATH:/home/petr/bin
export DAQ_PATH=/mnt/raid/daq
export DAQ_CONFIG_PATH=$DAQ_PATH/config/
export DAQ_LOG_PATH=$DAQ_PATH/log/
export DAQ_HOST=pcfcal02
export BEAM_DAQ_HOST=$DAQ_HOST
```

The directory /home/petr/bin on ATLROS-H6 contains a number of other useful scripts which can be run manually (for example, **`daqStatus)`** .

Input parameter: none
Return value: none. Currently, there is no return value to signal a success. It will be added later.

2.4.2 ***`int BeamCrate_t::DaqQuit(const string& wait)`***

Action: executes a “soft” DAQ stop, by sending a corresponding request. Effective only if the run state is *“Stopped”* (no run is ongoing).

Input parameter:

`wait`	“” or “nowait” – to return immediately, without waiting for a confirmation from DAQ; “wait” – to wait till the DAQ goes to *“DaqQuit”* (it does so just before exiting, after performing all necessary cleanup actions, and immediately before disconnecting from the dispatcher).

Return value:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected; `err_TIMEOUT` – time-out in waiting for the reply from DAQ (normally > one SPS cycle); `err_CONTEXT` – run is not stopped (wrong context).

2.4.3 Run control commands with the run context checking

`int BeamCrate_t::RunStop(const string& wait)`
`int BeamCrate_t::RunPause(const string& wait)`
`int BeamCrate_t::RunResume(const string& wait)`

Action:
These methods work by sending the corresponding requests to the beam DAQ. **`RunStop`** stops the ongoing run (valid in *“Running”* or *“Paused”* states); **`RunPause`** pauses a run (valid in *“Running”* state) and **`RunResume`** resumes the run (valid in *“Paused”* state).

Input parameter:

`wait`	“” or “nowait” – to return immediately, without any run context checking before and after sending the command; “wait” – to wait till the DAQ actually performs the required run action and modifies the trigger, accordingly. This mode is safer than “nowait”, but might involve some extra latency (see Section 2.1).

Return value:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected; `err_TIMEOUT` – time-out in waiting for the reply from DAQ (> one SPS cycle), `err_CONTEXT` – wrong run context.

2.4.4 ***`void BeamCrate_t::configure(RunPar_t par)`***

Action: sets the parameters for the next run, before starting the run. The method only fills the private member `run_par` of the class, without actually starting the run. No any validity checks are performed (they can be added, if needed). The **`GetRunPar`** method (next Section) can be used to retrieve the currently stored run parameters from the object.

Input parameter:

`par`	a variable of type RunPar_t (see section 1.1), containing a full set of run parameters. The old run parameters stored in the object will be overwritten.

Return value: none

2.4.5 ***`RunPar_t BeamCrate_t::GetRunPar(void)`***

Action: retrieves the currently stored run parameters from the object.

Input parameter: none
Return value:

a variable of type RunPar_t (see section 1.1), containing a full set of the stored run parameters. This method, like **`GetRunPar`**, involves no interaction with the beam DAQ.

2.4.5 ***`int RunStart`***

```
  int  RunStart(const RunPar_t& par);
  int  RunStart(void);
  int  RunStart(int rn);
  int  RunStart(int rn, int rt);
  int  RunStart(int rn, int rt, int rst);
```

Action: starts a run, by sending a corresponding request to the DAQ, with the parameters prepared according to the call option. Effective only if the run state is *"Stopped"* (no run is ongoing). Some elementary checks are performed on the run parameter values (in particular, it is checked that only one of `max_events` and `max_bursts` parameters is specified). The method returns immediately, without waiting for the *"Running"* status being acknowledged. If needed, use **`WaitFor`** method to ensure that the run has actually started.
Input parameters:

`par`	a variable of type RunPar_t (see section 1.1). The same meaning as for **`configure`** (Section 2.6).
`rn`	the run number
`rt`	the run type, (0=physics, 1=calibration, 2=special)
`rst`	the run sub-type, a non-negative value. There must be a one-to-one correspondence between possible run-type/run-subtype combinations and the configuration files in /raid/daq/config. Currently only a few configuration files are available, for test purposes (for example rt=0/rst=0). The configuration files include FEB, calibration board, CAMAC and trigger settings, as well as the read-out options. I will compile a full set of configuration files, once the run plan has been finalized.

The call without arguments uses the set of run parameters currently stored in the object, except that it increments the run number before sending the request to the DAQ.

Return value:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected; `err_RUNPAR` – error(s) in the run parameters; `err_CONTEXT` – wrong context: the current run status is not "Stopped".

2.4.6 ***`int BeamCrate_t::WaitFor(const string& wait)`***

Action: waits for receipt of the specified message from the beam DAQ. Useful for awaiting the "Running" state after starting a run. Other run control methods, like **`RunStop`**, **`RunPause`**, **`RunResume`** and **`DaqStop`** have a safe built-in option to wait for the corresponding run states. However, there is no restriction on using **`WaitFor`** anytime, to wait for any anticipated run context.

Input parameter:

`wait`	"" or "nowait" – to return immediately, without any waiting or context checking; Any of "Stopped", "Starting", "Running", "Paused", "Stopping", "DaqStopped" – see Section 2.1 for a discussion of the latency issues.

Return value:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected; `err_TIMEOUT` – time-out in waiting for the specified message **`wait`**.

2.4.7 Utility methods

`int BeamCrate_t::check_context(const string& expected_context)`

Action: compares the actual beam DAQ run context with the `expected_context`. See Section 2.1 for a discussion of the latency issues.

Input parameter:

`expected_context`	Any of "Stopped", "Starting", "Running", "Paused", "Stopping", "DaqStopped"

Return value:

`OK` – success; `err_CONN` – failed to subscribe to the dispatcher; `err_DAQ` – beam DAQ not connected; `err_TIMEOUT` – time-out in waiting for the for the context information from the DAQ; `err_CONTEXT` – the actual and the expected contexts do not match.

3. BeamData_t class (data receiver)

The class is intended for data receiver applications (saver, event builder). A BeamData_t object is almost always connected to the dispatcher and is subscribed to receive all events and run status messages. The main method **`get_rod`** automaticall disconnects only on occurrence of abnormal conditions or certain run contexts (usually, *"DaqQuit"* and *"Aborted"*). The available public methods are listed in Fig. 3.

3.1 Constructors

A BeamData_t object is created by one of two kinds of declaration similar to the BeamCrate_t constructors (Section 3.3):

```
BeamData_t(void);
BeamData_t(const string& host);
```

No connection is established or tried by the constructors.

3.2 Methods

3.2.1 `int BeamData_t::subscribe(void)`

Action: connects to the dispatcher with the subscription list requesting all events of all types (except BPC calibration events) and all run status reports from the beam DAQ.

Input parameter: none
Return value:

`OK` – success; `err_CONN` – failed to contact the dispatcher; `err_DAQ` – beam DAQ not connected.

3.2.2 `int BeamData_t::get_rod (unsigned long *buf, const int maxbuf)`

Action: fills the buffer `buf` with the next beam ROD fragment data. Alternatively, it returns a non-zero error code signalling a connection problem or a reception of an end-of-run condition (the DAQ reports *"Stopped"*). The existing connection is retained if data or end-of-run is received and dropped, otherwise. Important: the end-of-run condition is reported only once! [3]

After receiving an event fragment, some of its attributes (event number, type, size, integrity), as well as the corresponding run state attributes (run number, current run context), can be obtained by using of `ev_id`, `ev_type`, `check_CRC`, `run_num` and `RunStatus` methods.

Input parameters:

`buf` – a pointer to the external buffer to hold the event fragment;
`maxbuf` – the buffer size; the fragment is truncated if its size exceeds `maxbuf`;

[3] A subsequent call to **`get_rod`** will ignore the *"Stopped"* condition till the next data fragment arrives. This must be taken into account when implementing the receiver logic. If the receiver always knows in advance that the next run is going to be started, then a better solution would be to **`disconnect`** (Section 1.7) after the end-of-run interrupt and **`subscribe`** again during the configure-run phase. Leaving the BeamData_t object connected for a long time without calling **`get_rod`** might eventually block the dispatcher, if the beam DAQ keeps running. If the receiver is completely decoupled from the run control logic, then the best and the safest is to keep it always in **`get_rod`**.

Return value:

`OK` – success; `err_CONN` – connection to the dispatcher failed; `err_DAQ` – beam DAQ is not connected or was killed/aborted; `run_END` – end-of-run interrupt. The actual run context after the interrupt can be retrieved with **`RunStatus`** method (Section 1.7).

Example:

```
enum {MaxBuf=100000};
unsigned long buf[MaxBuf];
int rc;
BeamData_t dr;

dr.subscribe();

...
while(1) {
  rc=dr.get_ROD(buf,MaxBuf);
  switch(rc) {
  case BeamData_t::OK :
/* process the event */
   ...
   break;
  case BeamData_t::run_END :
/* process the end of run condition and return to get_rod
 ...
   break;
 default :
   goto error;
}
error: ...
```

3.2.3 Utility methods: event and run attributes

`int BeamData_t::ev_id(void)`
`int BeamData_t::ev_id(int& bc)`
`int BeamData_t::ev_id(int& bc, int& size)`
`int BeamData_t::ev_type(void)`
`int BeamData_t::run_num(void)`

Action: after a successful reception of an event fragment, return some event attributes stored in the ROD fragment header. Of course, the same values can be retrieved directly from the data.

Input parameters:

For `ev_id` with parameters, references to variable to hold the values for the bunch crossing ID (`bc`) and the fragment size (`size`) in full words.

Return value:

`ev_id` returns the event ID (sequential number of the event in the run); `ev_type` returns the event type (see [2] for details); `run_num` returns the run number.

4. The files

bc.sxw.pdf – this document
BeamDaq.h – the class declarations
dispatch.h – the header file for the ControlHost interface
libconthost.a – the ControlHost C-library
ControlHost.README – a mini-reference for the ControlHost package (C- and Fortran interfaces)
bc.c – a demo run controller
mkbc – a script to compile/link bc.c
dr.c – a demo data receiver
mkbc – a script to compile/link dr.c

The demo run controller code (bc.c) is a simple line-mode commander illustrating the usage of the BeamCrate_t class. The program dr.c illustrates the usage of BeamData_t class. Both programs have been tested on ATLROS-H6 machine, under the account petr@atlros-h6.

References

[1] http://www.nikhef.nl/~ruud/HTML/choo_manual.html (decribes a dialect of ControlHost, but contains a copy of the C-library manual corresponding to the version I am using).

[2] P.Gorbunov, "H6 Beam Crate ROD fragment format". The latest version can be found in /afs/cern.ch/user/p/petr/public/Eformat directory.

Figures

```
void debug(const int& level);    // sets the degig level (0, 1, 2)
void disconnect(void);           // disconnects from the beam host

int  check(const int& print);    // checks the connection status
int  RunStatus(string& status); // current run status
```

Figure 1: A summary of common BeamCrate_t and BeamDaq_t public methods.

```
check_context(const string& context)        // a run context checking

void DaqRun(void);                          // beam DAQ control
void DaqQuit(void);

void configure(RunPar_t par);               // run control
int  RunStart(const RunPar_t& par);
int  RunStart(void);
int  RunStart(int rn);
int  RunStart(int rn, int rt);
int  RunStart(int rn, int rt, int rst);
RunPar_t GetRunPar(void);

int  RunStop(const string& wait);
int  RunPause(const string& wait);
int  RunResume(const string& wait);

int  WaitFor(const string& wait_for);
```

Figure 2: A summary of BeatCrate_t public methods.

```
int  subscribe(void)            // to connect to the beam DAQ

int  get_rod(unsigned long *buf, const int maxbuf)

int  check_CRC(void);           // not yet available
int  ev_type(void);             // event type
int  run_num(void);             // run number

int  ev_id(void);               // event number and, optionally,
int  ev_id(int& bc);            // the bunch crossing ID and
int  ev_id(int& bc, int&siz);   // the fragment size
```

Figure 3: A summary of BeatData_t public methods.