CALICE DAQ User and Reference Manual

P. D. Dauncey*

Department of Physics, Blackett Laboratory,
Imperial College London,
Prince Consort Road,
London, SW7 2AZ, UK.

August 23, 2005

*Email: `P.Dauncey@imperial.ac.uk`

Contents

1 Introduction

For most users (doing analysis directly from the raw data or working on the LCIO conversion) then the section on offline use if most useful.

2 Basic concepts

The DAQ system is mainly based around the CALICE Readout Card (CRC, pronounced "Kirk").

The data are based on the concept of records, which are generalised events. Each record indicates an occurance within the DAQ system. Each process within the DAQ can add data to the record (but not remove anything). The data within the record are organised into subrecords, each of which has a well-defined structure. The records are manipulated as contiguous chunks of memory with regard to being copied, transported, etc.

The code is intended to be light-weight and transportable (meaning to different sites). Having said that, it is written in C++ and is only intended to work with Linux. The only external code needed is ROOT (for both online and offline applications) and HAL and a HAL-supported VME driver (for online only). These are included as soft links into the directory structure, allowing them to be installed wherever convenient on the computer being used.

2.1 Records

Each record contains a short header with the most basic information which is then (optionally) followed in memory by an arbitrary amount of data. The header contains just the time and type of the event and the number of bytes of data following, for which zero (i.e. no data) is valid. The organisation of the following data is not defined by the record classes at all; it can be any structure desired. Because the header size and length of the following data is known, the records can be manipulated easily without any further knowledge. Hence, the I/O and transport classes for records do not need to depend on the data structure within the record.

The internal record structure actually used for CALICE involved subrecords, which are well-defined chunks of data inserted into the record. Each subrecord starts with a subheader, which contains an identifier (to indicate the data contents) and the number of bytes (or words?) of the data in the subrecord. This allows many manipulations of the subrecords to be done without knowledge of the actual data in the subrecord. However, to actually use the subrecords, then the identifier must be checked to ensure the data are as expected. This is done by associating each identifier used with a data storage class which defined the actual subrecord data structure for that identifier. Hence, a list of identifiers for each class must be maintained and whenever a class is modified in a non-backwards compatible way, a new identifier must be assigned to the new class.

These data storage classes must lie completely within the contiguous data of the subrecord and this puts limitations on the classes used. They must contain all their data within the object or contiguously following the object, which means no pointers can be used. In addition, although inheritance can be used, no virtual functions are possible. To manipulate these storage objects, an inserter or accessor class is used to add or read the objects, respectively.

2.2 Online Dataflow

The movement of the records within the DAQ system is done using the DIO system.

The VME readout is based on the HAL system.

2.3 Offline analysis

The actual "final" analysis of the data will not be done from the raw data as written out by the DAQ but from a reduced dataset. This will be in LCIO format and will only contain calorimeter channels which are above a threshold. Their values will be calibrated and pedestal subtracted in these files.

Hence, the meaning of "offline" here is therefore the process needed to take the raw data files (defined in terms of records) and do any analysis which requires the raw data. The most obvious case is the actual job which does the data reduction to the LCIO dataset. Another obvious case is the calculation of the calibration constants (pedestals, gains, etc) used in the LCIO conversion job. However, it is likely all calibration run analysis will also be done from the raw data as will all quick checks of data quality. These are the tasks which are covered by the concept of offline here.

2.4 Conventions

All C++ header files are .hh, all main programs are .cc.

The subdetectors have abbreviations emc, ahc, dhc, trg, slw, bml for ECAL, AHCAL, DHCAL, trigger, slow controls and beam line, respectively.

3 Hardware

3.1 CRC overview

The CRC is a (substantially modified) adaptation of the CMS silicon tracker readout board, the Front End Driver (FED). However, the whole of the front section of the board was redesigned for CALICE use and the firmware for the FPGAs has also been modified. An overview of the CRC is shown in figure 1.

The CRC contains eleven Xlinix Virtex-II FPGAs, running three firmware designs. The board is mainly controlled by the Back End (BE) FPGA. This communicates with the VME controller via the VME FPGA and with the Very Front End (VFE) electronics via the Front End (FE) FPGAs. Each CRC has one VME, one BE and eight FE FPGAs; all the FE FPGAs run the same firmware.

There are two data paths via VME into the CRC. One is a serial I/O link, which is used for configuration and slow data readout; this has both read and write access. The second path is a read-only fast data path, the "Vlink", which allows VME block transfers.

The BE distributes configuration data and the trigger to the FEs and collects readout data from the FEs. These data are stored in a 8MByte Quad Data Rate (QDR) memory on the CRC. This can be read via the Vlink.

3.1.1 ADCs and DACs

The main function of the CRCs is to digitise the analogue data stored in the VFE electronics. The analogue level is stored using a sample-and-hold and then subsequently clocked out on a multiplexed line to the CRC. These analogue levels are input to six dual ADCs per FE and the digitised data are buffered in the FE before being transfered later to the BE.

The ADCs used are TI ADS8361 500 kHz, 16 bit devices. Each component contains two ADCs, each of which has two inputs; one is used for the VFE analogue signal and the other for a direct input from the on-board DACs. Which input is to be digitised is controlled by the FE and configurable via software.

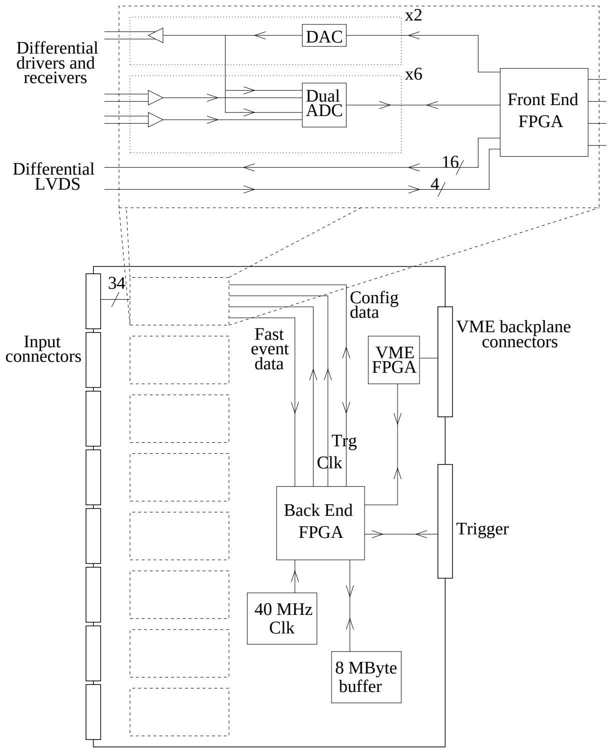

Figure 1: Overview of CRC

The DACs used are AD AD5541 16 bit devices. There are two DACs per FE, one for each connector.

3.1.2 Trigger distribution

The trigger is handled through a standard CRC. The trigger input signals are input via the pins on the VME J2 connector. The logic and control of the trigger in implemented in the BE and, following a trigger decision, the trigger is output to the J0 connector via a 1-to-20 fanout. The outputs from the fanout are then distributed via cables or a custom J0 backplane to the other CRCs.

3.1.3 VFE acquisition sequence

The signal timing sequence to drive the data acquisition sequence for the VFE electronics is shown in figure 2. This sequence is run through for every trigger.

All signals are LVDS except OUTPUTs and VCALIBs.

3.1.4 Connector signals

The signals on the connector XXX.

DIAGRAM OF CONNECTOR AND PIN NUMBERING

DIAGRAM OF CONNECTOR PINS

3.1.5 FE acquisition sequence

In addition to the VFE signals, the FE needs to generate the signals needed within the CRC with the correct timing. These signals consist of the ADC control and the FE-to-BE data transfer

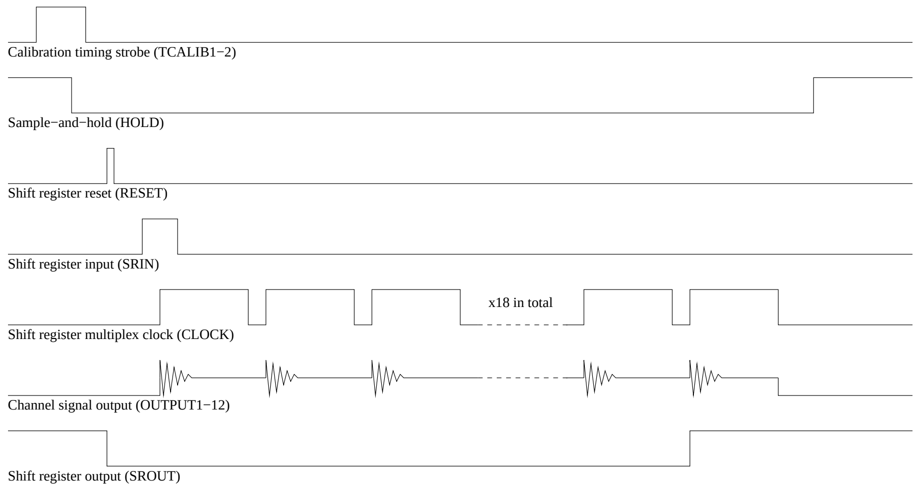

Figure 2: VFE acquisition sequence

signals.

The digitised values are stored in a FIFO implemented within the FE FPGA. These data are transfered to the BE and subsequently to the QDR memory after the FE sequence is complete.

3.1.6 FE-to-BE data transfer

The falling edge of the HOLD signal starts the FE-to-BE data transfer. The transfer involves four data packets, two in each direction. The FE initiates the first, the FrameSyncOut, following the HOLD falling edge and this consists of one leading bit followed by 80 bits of data; these are currently fixed to the values 0x0102030405060708090a.

3.1.7 BE, VME access, serial I/O and Vlink

The main data transfer method off the CRC is the Vlink. This reads the data stored in the QDR memory through VME.

All other read, and all write, access to the CRC is via the serial I/O link.

3.1.8 Monitoring data

NS LM82 and power shutdown. AD ADM1025A.

3.2 CRC hardware setup

3.2.1 Jumper array

The CRC can handle several types of VFE electronics by setting jumpers in the large jumper array immediately behind the FE connectors. The ECAL VFE electronics has three types; full, left and right, while the AHCAL VFE electronics has two; left and right. In addition, the CRC is designed to accomodate a fully-digital interface to another system if this is ever required.

REQUIREMENTS FOR FULL, LEFT AND RIGHT.

Link array has 15 connections; 12 ADCs, 2 DACs and 1 detection.

See figure 3. Link array and VFE types, plus readback

VCALIB shared if full, two separate VCALIBs if left and right?

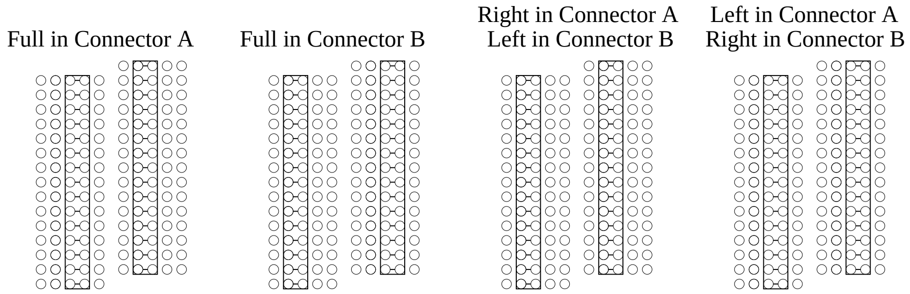

Figure 3: Link array settings, views from the top (component) side of the CRC with the FE connectors to the left.

3.2.2 CompactFlash card

Location of reset button

Xilinx SystemACE

Reload procedure and power up issues

How to make a new CompactFlash card

3.2.3 EPROM

Each CRC has a non-volatile EPROM which can contain long-term data. This is a Xicor X24165 2k $\times 8$ bit memory device which can be locked to be read-only using a jumper on the CRC.

This is used to store the CRC unique serial number.

Serial number convention.

Location of write jumper; diagram and photo?

3.2.4 CRC LED indicators

3.3 CRC software configuration

3.3.1 FE sequence timing

3.3.2 VFE enables

3.3.3 BE

3.3.4 AHCAL VFE shift registers

3.4 CRC trigger

Note that the trigger busy is implemented by holding it high.

3.4.1 BE-Trg

3.4.2 CRC trigger fanout

3.4.3 Backplane trigger distribution

4 Online structure

4.1 State machine

4.2 Records as transitions

4.3 Data groups

4.3.1 Run, R/O

4.3.2 Configuration, R/W

4.3.3 Event, R/O

4.3.4 Trigger control

4.4 XxxConfiguration and XxxReadout

4.5 Other code

4.5.1 HAL

4.5.2 SBS

4.5.3 ROOT

5 Offline structure

There are generally two ways to use the raw data files offline. One uses the minimal amount of DAQ software and requires the overall infrastructure to be supplied by the user; for example, this is the system used for the LCIO conversion. This minimal software is contained in the "records" package and only this is needed.

The second method is to work within the DAQ offline structure as a DAQ "user", for which not only the records package but also the "daquser" package are needed. In addition, the correct directory structure needs to be used.

Both methods are described below. The use of the records package for the first method is identical for both, so this is decribed first.

5.1 Use of the records package

5.1.1 Records as data containers

5.1.2 Accessing the basic record information

The code for handling records is in `records/inc/rcd`. Normally, a record is handled within some specific array of memory, defined by the `RcdArena` class. This inherits from the basic record class `RcdRecord`.

The header of each record is encapsulated in a class `RcdHeader`, from which `RcdRecord` inherits, and the information stored is accessed though methods of this class. These are

- `RcdHeader::recordTime()`: Returns a `UtlTime` object containing the timestamp for the record.

- `RcdHeader::recordType()`: Returns a `RcdHeader::RecordType` enum value giving the type of record; start of run, end of run, event, etc.

Both these items can be printed using the `print` method, e.g.

```
arena.RcdHeader::print(std::cout);
```

Since all other manipulations of data are done through subrecords, then the other methods of records are not normally required.

5.1.3 Reading and writing records

To read a record from a file, then a `RcdReader` object is used. This is also found in the subpackage `records/inc/rcd`. Both binary (`.bin`) and ascii (`.asc`) record files can be handled. The `RcdReader` object should be used to open, read and close the record file as follows, for example for a binary file:

```
RcdArena arena;
RcdReaderBin reader;
assert(reader.open("InputFile"));  // ".bin" is appended

while(reader.read(arena)) {
  // Use record
}

assert(reader.close());
```

In a similar way, a `RcdWriter` object can be used to write records to a file, again in either binary or ascii format. The method interface is very similar to that for reading

```
RcdArena arena;
RcdReaderBin reader;
RcdWriterBin writer;
assert(reader.open("InputFile"));  // ".bin" is appended
assert(writer.open("OutputFile")); // ".bin" is appended

while(reader.read(arena)) {
  // Use record
  assert(writer.write(arena));
}

assert(reader.close());
assert(writer.close());
```

5.1.4 Accessing subrecords within records

The code for handling subrecords is in `records/inc/sub`. Only classes which have been registered in `records/inc/sub/SubRecordTypes.hh` can be used as subrecords and hence stored within records. Therefore, this file by definition contains the definitive list of subrecords.

To access a particular set of subrecords, then a subrecord accessor `SubAccessor` is used. This can return a `std::vector` list of pointers to all the occurances of any subrecord in a record. The code would look similar to:

```
SubAccessor accessor(arena);
std::vector<const DaqRunStart*> v(accessor.access<DaqRunStart>());
```

which gets a list of pointers to `DaqRunStart` objects, for example.

5.1.5 Fixing subrecords

As the DAQ software develops, the subrecord classes will be modified, not necessarily in a backwards-compatible way. To handle this, each subrecord class has a version number and conversion constructors are supplied to change older versions to the latest version, when possible. The `records/inc/fix` subpackage contains code to perform these conversions on-the-fly and can be used as follows:

```
FixAll fix;
...
// For each record
fix.fix(arena);
```

The `records/src` subdirectory contains one main program, `example.cc` which simply puts some of the above pieces together.

```
#include "RcdArena.hh"
#include "RcdReaderBin.hh"
#include "RcdReaderAsc.hh"
#include "RcdWriterBin.hh"
#include "FixAll.hh"

using namespace std;

int main(int argc, const char **argv) {
  if(argc<2 || argc>3) {
    cout << "Usage: " << argv[0]
 << " <input filename> [<output filename>]" << endl;
    return 1;
  }

  // Memory array for records
  RcdArena arena;

  // Record reader; bin or asc
  RcdReaderBin reader;
  //RcdReaderAsc reader;

  // Record writer if required
  RcdWriter *writer(0);
  if(argc==3) writer=new RcdWriterBin;

  // Fixers to repair data
  FixAll fixAll(4); // 4 prints up to and including configuration

  // Open files
  if(!reader.open(argv[1])) return 2;
```

```
  if(argc==3 && !writer->open(argv[2])) return 3;

  // Read records until end of file
  while(reader.read(arena)) {
    fixAll.fix(arena);
    if(argc==3) writer->write(arena);
  }

  // Close files
  if(!reader.close()) return 4;
  if(argc==3 && !writer->close()) return 5;

  return 0;
}
```

5.2 Use of the daquser package

5.2.1 User directory

To use the software, the user needs to create a directory with a specific structure. This has the following subdirectories:

- `inc`: Contains user classes.
- `src`: Contains user main programs.
- `bin`: Contains the executables made from the `src` code.

In addition, it needs either subdirectories or more usually soft links as follows:

- `records`: The above central software `records` directory.
- `daquser`: The above central software `daquser` directory.
- `data`: Contains the data directories (see below).

For online use, the directory needs to have

- `online`: The above central software `online` directory.
- `sbs`: A soft link to the top of the SBS driver directory.
- `hal`: A soft link to the top of the HAL directory (online only).

5.2.2 bin/build, PATH, setenv

Within the `bin` subdirectory, a link should be created to the `daquser` scripts `bin/build` Then, `build xxx` will compile and link the file `xxx.cc` and write the resulting executable to `bin/xxx`. The file `xxx.cc` is searched for (in order) in `src`, `daquser/src` and finally `records/src`.

5.3 Data directories

5.3.1 dat

5.3.2 slw

5.3.3 log

5.3.4 How to access subrecords

5.4 Overview of subrecords

5.4.1 Run, R/O

5.4.2 Configuration, R/W

5.4.3 Event, R/O

5.4.4 Trigger control

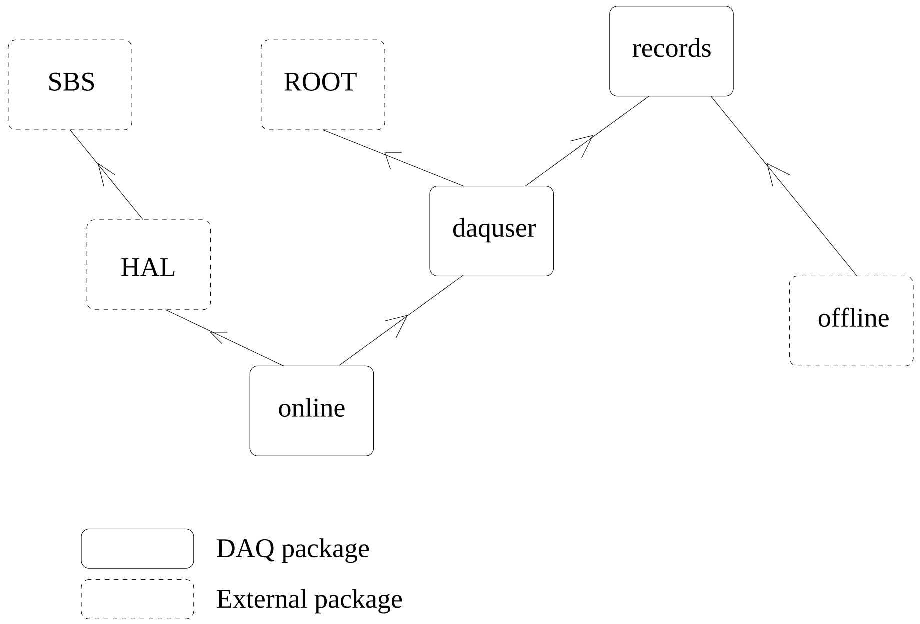

Figure 4: Conceptual overview of the DAQ and external software packages.

Appendix A: Code organisation

An overview of the three software packages is shown in figure 4.

5.5 DAQ software

The DAQ software is organised into three main directories:

- `online`: This contains code which is only used in the online, i.e. the actual DAQ process.
- `daquser`: This contains code which is used both online and daquser, during read back or analysis of the data.
- `records`: This contains code which needs to be used for the LCIO conversion.

For daquser use, only the `daquser` and `records` directories need to be installed although ROOT must also be available. For online use, all three directories need to be installed together with ROOT, HAL and the SBS VME driver.

The `records` directory only contains subdirectories `inc`, `src`, `bin` and `doc`. `inc` contains the classes.

- `inc`: Contains the classes arranged in further subdirectories.
- `src`: Contains the main programs.
- `bin`: Contains scripts to build executables from the `src` code and other useful scripts.

For the `daquser` directory, the subdirectories are:

- `inc`: Contains the classes arranged in further subdirectories.
- `src`: Contains the main programs.
- `bin`: Contains scripts to build executables from the `src` code and other useful scripts.

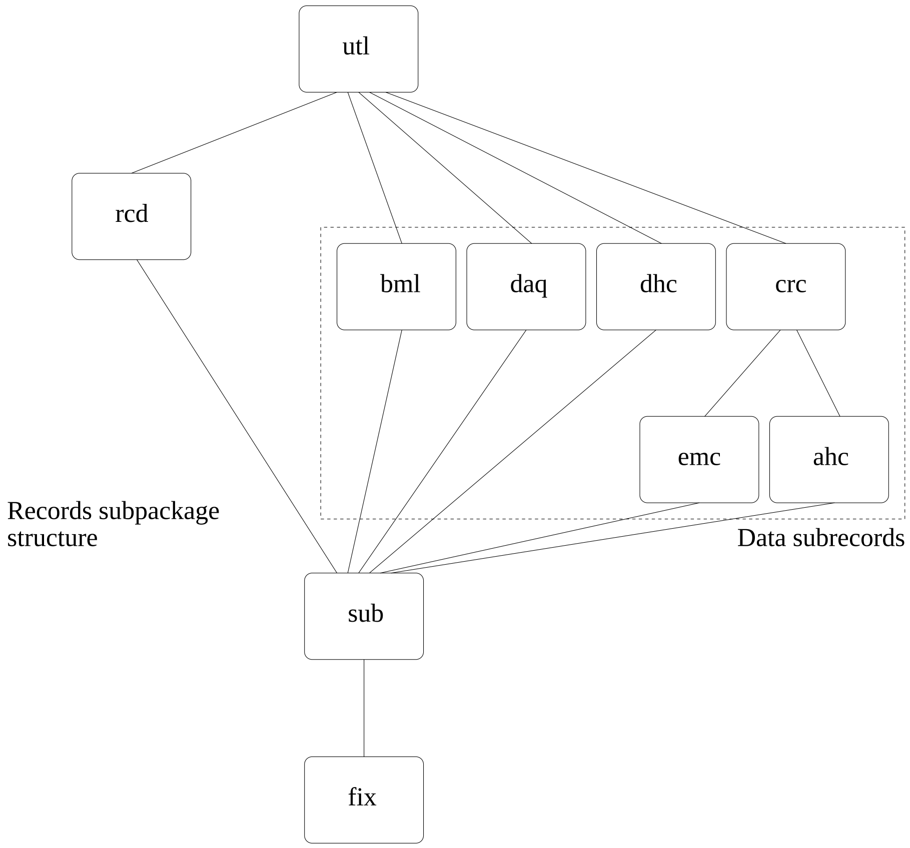

Figure 5: Conceptual overview of the records subpackages.

The `online` subdirectories are similar to the ones in `daquser` but have a fourth subdirectory, `hal`, which contains the VME address space map definition files for the VME modules used online.

5.6 Records software

An overview of the subpackages is shown in figure 5. No class is able to know about or use any class in a lower group.

The classes in the `records/inc` subdirectory are organised into two different groups. The first contains classes used to manipulate data and the second contains classes to store and handle the data of the subrecords. These are split into subdirectories as follows; for the first group

- `utl`: Contains utility classes.
- `rcd`: Contains the record structure and basic I/O classes.
- `sub`: Contains the subrecord organisation classes.
- `fix`: Contains the subrecord fixing code.

and for the second group

- `daq`, `crc`, `emc`, `ahc`, `dhc`, `trg`, `slw`, `bml`: Contains the subrecord data storage classes; the generic CRC data and the various subdetectors.

5.6.1 Classes in `inc/utl`

The utility classes perform various commonly needed functions. `UtlPrintHex` prints values in decimal, hex and (optionally) binary for debugging purposes. `UtlLocation` contains the crate, VME slot, component (and an extra label) for any data. `UtlPack` `UtlTime` `UtlTimeDifference`

5.6.2 Classes in `inc/rcd`

These classes

5.6.3 Classes in `inc/sub`

5.6.4 Classes in `inc/fix`

DAQUSER

5.6.5 Classes in `inc/daq`

5.6.6 Classes in `inc/crc`

5.6.7 Classes in `inc/emc`

5.6.8 Classes in `inc/ahc`

5.6.9 Classes in `inc/dhc`

5.6.10 Classes in `inc/trg`

5.6.11 Classes in `inc/slw`

5.6.12 Classes in `inc/blm`

5.6.13 Classes in `inc/chk`

5.6.14 Classes in `inc/hst`

5.6.15 Classes in `inc/dsp`

5.6.16 Classes in `inc/skt`

5.6.17 Classes in `inc/shm`

`UtlAverage` `UtlRollingAverage` `UtlArguments`

5.7 Online software

An overview of the subpackages is shown in figure 6.

- `hal`: Contains the code for
- `dio`: Contains the code for data transport.
- `onl`: Contains the code for online use of the `dio` classes.
- `shm`, `skt`: Contains classes to handle data for shared memory and sockets, respectively.
- `sem`, `sgn`, `shm`, `skt`, `pip`: Contains classes to allow the use of inter-process connunication, namely semaphores, signals, shared memory, sockets and pipes, respectively.
- `vme`: Contains the generic VME classes.

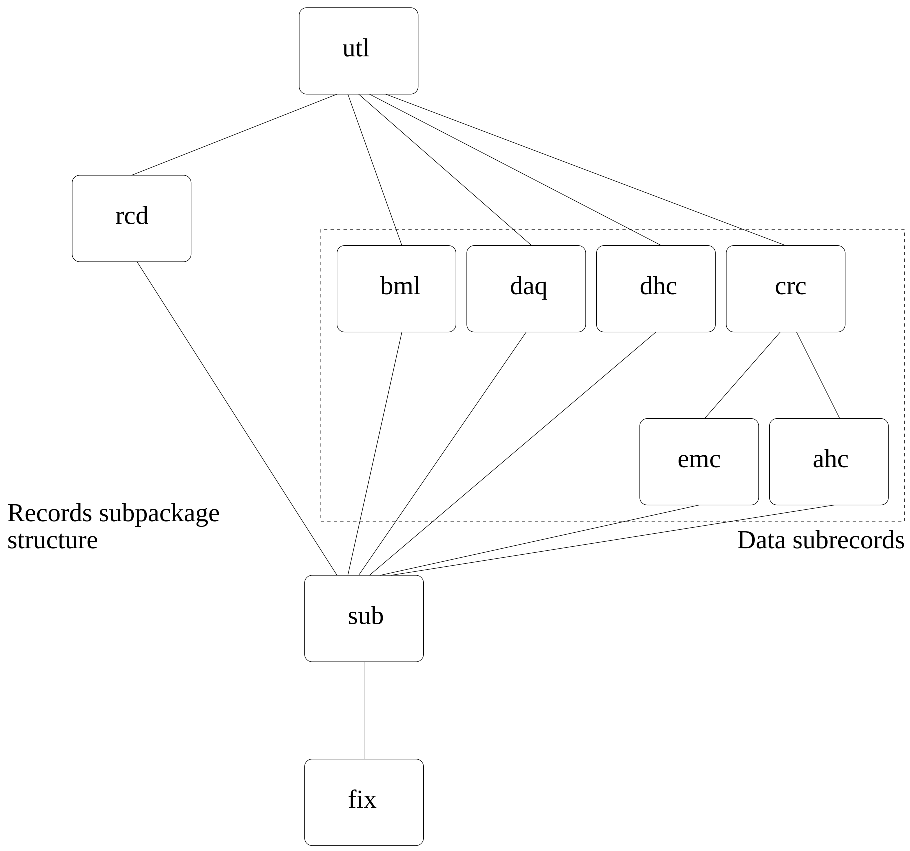

Figure 6: Conceptual overview of the online subpackages.

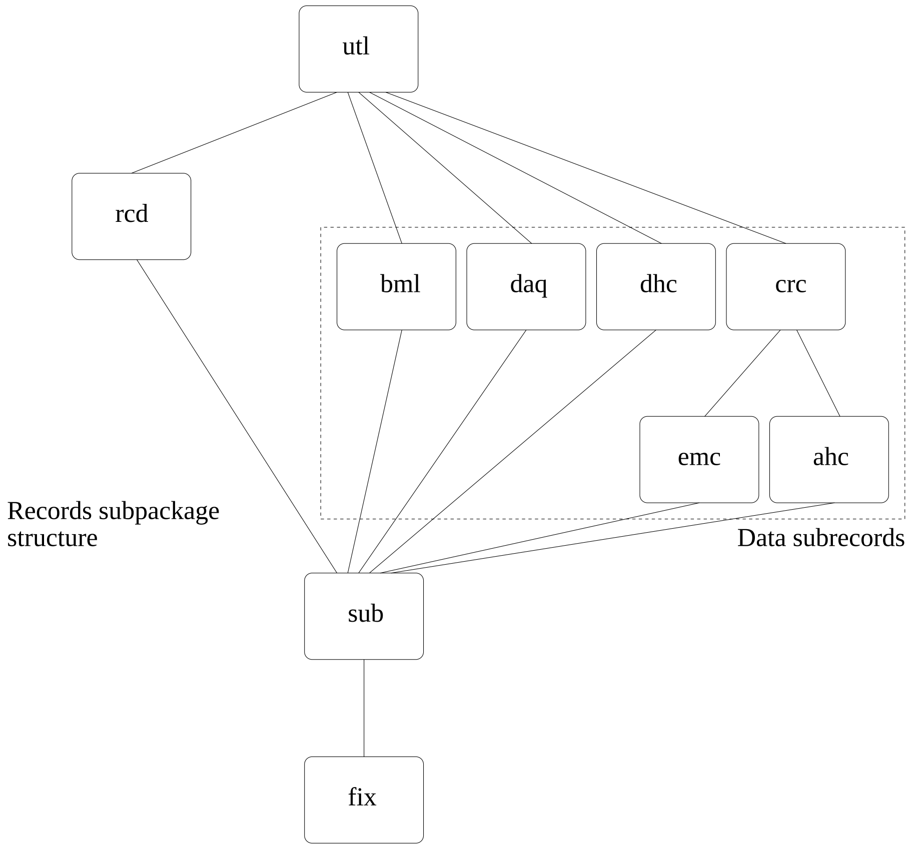

Figure 7: Conceptual overview of the offline subpackages.

- `emc`, `ahc`, `dhc`, `trg`, `slw`, `bml`: Contains the subdetector-specific online classes.
- `chk`, `hst`, `dsp`: Contains low level checking, histogramming and display classes for data analysis.

5.7.1 Classes in `inc/dio`

5.7.2 Classes in `inc/onl`

5.7.3 Classes in `inc/vme`

5.7.4 Main programs in `src`

5.8 Offline software

An overview of the subpackages is shown in figure 7.

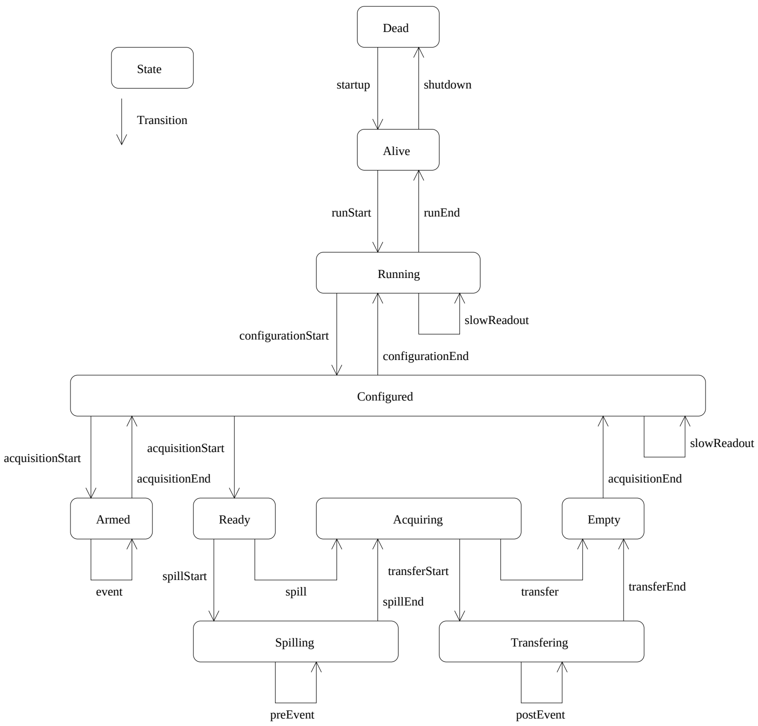

Figure 8: The DAQ state machine

5.8.1 Classes in `inc/cal`

5.8.2 Classes in `inc/lco`

5.8.3 Main programs in `src`

Appendix B: State machine

5.9 Definitions and transitions of states

The DAQ state machine is shown in figure 8.

- State Dead: no DAQ program running
  - Transition startUp to state Alive: default
- State Alive: program initialised but no run in progress
  - Transition runStart to state Running: default
  - Transition shutdown to state Dead: when finished number of runs or on user command
- State Running: Run initialised
  - Transition configurationStart to state Configured: default
  - Transition slowReadout to state Running: default if monitoring run - Read out all slow data. Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.
  - Transition runEnd to state Alive - on user command - Same as for RUNSTART

- State Configured: All configuration data loaded. In this state, need to have - Trigger in BUSY state - All QDR buffers empty - All event counters cleared
  - Transition acquisitionStart to state Armed: default if XXX run
  - Transition acquisitionStart to state Ready: default if YYY run
  - Transition slowReadout to state Configured: if more than minimum time since last slowReadout
  - Transition configurationEnd to state Running
- State Armed: Ready to take event data
  - Transition event to state Armed: default
  - Transition acquisitionEnd to state Configured
- State Ready: prepared for spill
  - Transition spillStart to state Spilling
  - Transition spill to state Acquiring
- State Spilling: taking triggers during spill
  - Transition preEvent to state Spilling: default - Release central trigger busy. S/w trigger or poll for trigger. Read out pre-event data.
  - Transition spillEnd to state Acquiring
- State Acquiring: taking data during spill
  - Transition transferStart to state Transfering
  - Transition transfer to state Empty
- State Transfering: reading out buffered data taken during spill
  - Transition postEvent to state Transfering: default if not finished all event data
  - Transition transferEnd to state Empty: when finished number of events
- State Empty: finished data acquisition for a spill
  - Transition acquisitionEnd to state Configured

  - Transition CONFIGURATIONEND to state ALIVE - - Readback all configuration data
  - Transition CONFIGURATIONSTOP to state ALIVE - when output file size too big, or user command - Readback all configuration data
  - Transition SLOWREADOUT to state CONFIGURED - occasionally - Read out all slow data. Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.

  * SM needs to know if finished events, file size, user command and time since last SLOWREADOUT

- State SETUP = 3

- Transition CONFIGURATIONSTART to state CONFIGURED - default - Write and readback configuration data. Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.
- Transition RUNEND to state ALIVE - when finished number of configurations - Same as for RUNSTART plus VFE load/flush
- Transition RUNSTOP to state ALIVE - when output file size too big, or on user command - Same as for RUNEND

* SM needs to know if finished configs, file size and user command

- State TRIGGERED = 5
  - Transition EVENT to state CONFIGURED - default - Read event data

5.9.1 STATE MACHINE 2 = SPILL WITH PRE-EVENT AND EVENT

- State SETUP = 3
  - Transition CONFIGURATIONSTART to state CONFIGURED - default - Write and readback configuration data Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.
  - Transition RUNEND to state ALIVE - when finished number of configurations - Same as for RUNSTART plus VFE load/flush
  - Transition RUNSTOP to state ALIVE - when output file size too big, or on user command - Same as for RUNEND

  * SM needs to know if finished configs, file size and user command

- State CONFIGURED = 4 In this state, need to have - Trigger in BUSY state - All QDR buffers empty - All event counters cleared
  - Transition STARTSPILL to state SPILLED - default - S/w spill or poll for spill
  - Transition CONFIGURATIONEND to state ALIVE - when finished number of spills or events - Readback all configuration data
  - Transition CONFIGURATIONSTOP to state ALIVE - when output file size too big, or user command - Readback all configuration data

  * SM needs to know if finished spills or events, file size and user command

- State SPILLED = 5
  - Transition PREEVENT to state SPILLED - default - Release central trigger busy. S/w trigger or poll for trigger (must all osc or timeout) Read out pre-event data, inc spill and buffer status
  - Transition SPILLEND to state READQDR - after spill end, full buffer or finished number of events - S/w unspill or poll for spill end
  - Transition SPILLSTOP to state READQDR - when output file size too big, or user command - Same as SPILLEND

  * SM needs to know if spill status, if finished events or full buffer, file size and user command

- State READQDR = 6
  - Transition EVENT to state READQDR - if not finished events - Read event data
  - Transition SLOWREADOUT to state CONFIGURED - after last event - Read out all slow data Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.

  * SM needs to know if finished events

5.9.2 STATE MACHINE 3 = SPILL WITH EVENT ONLY

- State SETUP = 3
  - Transition CONFIGURATIONSTART to state CONFIGURED - default - Write and readback configuration data Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.
  - Transition RUNEND to state ALIVE - when finished number of configurations - Same as for RUNSTART plus VFE load/flush
  - Transition RUNSTOP to state ALIVE - when output file size too big, or on user command - Same as for RUNEND

  * SM needs to know if finished configs, file size and user command

- State CONFIGURED = 4 In this state, need to have - Trigger in BUSY state - All QDR buffers empty - All event counters cleared
  - Transition STARTSPILL to state SPILLED - default - Enable trigger timeout (and release if needed) S/w spill or poll for spill or number of events
  - Transition CONFIGURATIONEND to state ALIVE - when finished number of spills or events - Readback all configuration data
  - Transition CONFIGURATIONSTOP to state ALIVE - when output file size too big, or user command - Same as CONFIGURATIONEND

  * SM needs to know if finished spills or events, file size and user command

- State SPILLED = 5
  - Transition SPILLEND to state READQDR - default - Multiple S/w trigger. Poll for spill end, finished number of events, full QDR or user command. Or s/w unspill, poll for finished number of events, full QDR or user command. Readout pre-evet data Unset trigger timeout.

- State READQDR = 6
  - Transition EVENT to state READQDR - - Read event data
  - Transition SLOWREADOUT to state CONFIGURED - after last event - Read out all slow data Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.

  * SM needs to know if finished events

5.10 Definitions of transitions

- Transition startUp
  - Initialise all CrcVmeDevices, reset (with care), load AHCAL VFE DAC settings (but not whenever the program is started)
- Transition shutdown
  - Readback AHCAL VFE DAC settings (but not whenever the program is stopped)
- Transition runStart
  - Readback Crc run info and slow run info. Read EMC stage (if not slow)
  - Set and readback slow control settings
- Transition runEnd
  - Readback Crc run info and slow run info. Read EMC stage (if not slow)
  - Pptional VFE load/flush
  - Readback slow control settings
- Transition runStop (obsolete)
  - Same as for runEnd
- Transition configurationStart
  - Write and readback configuration data. Ensure trigger timeout is off and send s/w central trigger. Flush all buffers and counters.
- Transition configurationEnd
  - Readback all configuration data
- Transition configurationStop (obsolete)
  - Same as configurationEnd
- Transition slowReadout
  - Read out all slow data.
- Transition acquisitionStart
  -
- Transition acquisitionEnd
  -
- Transition spillStart
  - Enable trigger timeout (and release if needed) for SM3 S/w spill or poll for spill, number of events, QDR full
- Transition spillEnd

  - Poll for spill end, finished number of events, full QDR or user command. Or s/w unspill, poll for finished number of events, full QDR or user command.
  - Multiple S/w trigger. Poll for spill end, finished number of events, full QDR or user command. Or s/w unspill, poll for finished number of events, full QDR or user command. Readout pre-event data? Unset trigger timeout for SM3

- Transition spill
  - Poll for spill end, finished number of events, full QDR or user command. Or s/w unspill, poll for finished number of events, full QDR or user command. Poll for spill end, finished number of events, full QDR or user command. Or s/w unspill, poll for finished number of events, full QDR or user command. S/w trigger or poll for trigger (must allow osc or timeout for SM2) Read out pre-event data, inc spill and buffer status

- Transition transferStart
  -

- Transition transferEnd
  -

- Transition transfer
  - Read all QDR buffer data

- Transition preEvent
  - Release central trigger busy. S/w trigger or poll for trigger (must allow osc or timeout for SM2) Read out pre-event data, inc spill and buffer status

- Transition postEvent
  - Read out one event from QDR buffers

- Transition event
  - Same as preEvent and postEvent