T2K TPC – DAPNIA

DCC KIT DAQ USER MANUAL

Shebli Anvar, Dapnia/Sédi/Lilas

T: +33 1 69 08 78 32
M: +33 6 63 31 92 26
@: Shebli.Anvar@cea.fr

Version 1.0

1. INTRODUCTION

This document describes the use, the formatting of the data files produced by the DCC Kit Daq (DKD) software and the programmatic way to access configuration parameters. It is needed for the proper analysis of all data acquired with the DKD. A first section is briefly describes how the DKD interface relates to data files and configuration; the second section is dedicated to the data files format. The third section describes the C++ API that allows to access configuration parameters.

2. DKD INTERFACE

Figure 1 is a screen shot of the DKD graphical user interface (GUI) :

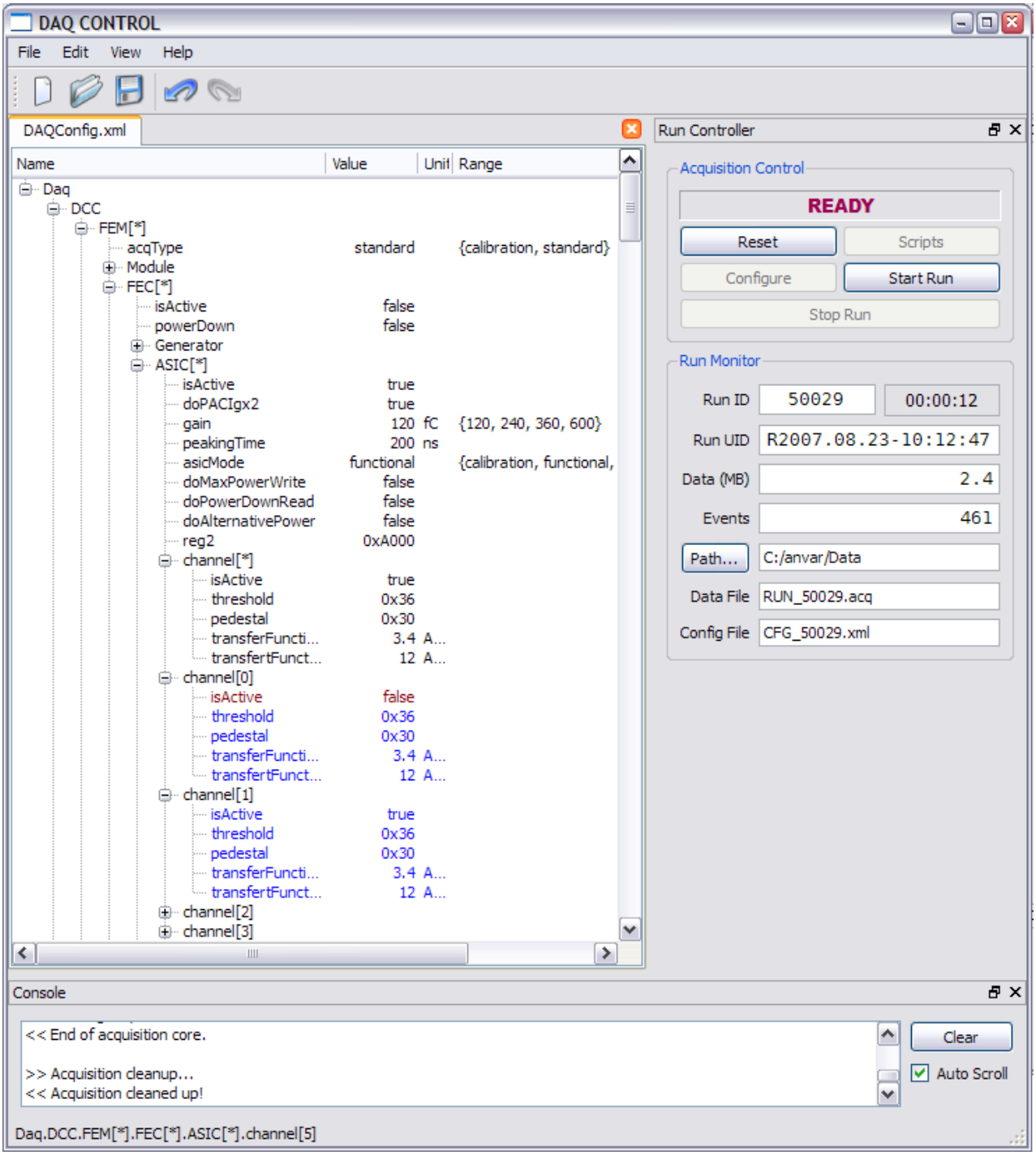


Figure 1: DCC Kit Daq (DKD) Graphical User Interface

The Configuration Panel

The function of the "tree-like" left panel is to display and edit the configuration parameters of the run. The exact logic, semantics and use of the configuration will be fully described in a future document. The main features of the configuration is as follows :

- A configuration is a tree of objects, sub-objects, etc. until you reach leaf values which represent the actual parameters of the run.
- Parameters (leaf values) can be of 5 types : integer (signed), hexadecimal (unsigned), real (double), boolean and string.
- A parameter is specified by a path in the tree, a name and an optional index. For instance, you may see on the screen shot the string parameter
  `Daq.DCC.Fem[*].acqType`
  having the value "`standard`", or the integer parameter
  `Daq.DCC.Fem[*].FEC[*].ASIC[*].gain`
  having the value "`120`".
- A parameter can have an optional unit which has no other incidence than adding some human-readable information (see examples on screen shot). It can also have a range that restricts the set of values it can take. Ranges are either intervals like `[0x0, 0x1FF]` or enumerations like `{120, 240, 360, 600}`.
- The '`*`' index indicates default values that are inherited by objects with the same path using specific indexes. For instance, the `Daq.DCC.Fem[*]` object contains default values for all `Daq.DCC.Fem` objects. Naturally, a default value or object can be overridden with specific values in specific instances. A value inherited from a default object is displayed in blue. An overridden value is displayed in red.

The Acquisition Control Panel

The acquisition logic follows a "state machine" represented in Figure 2, where rounded rectangles represent the Daq states and the arrows represent the transitions that occur when the specified 'event' (arrow label) is issued:

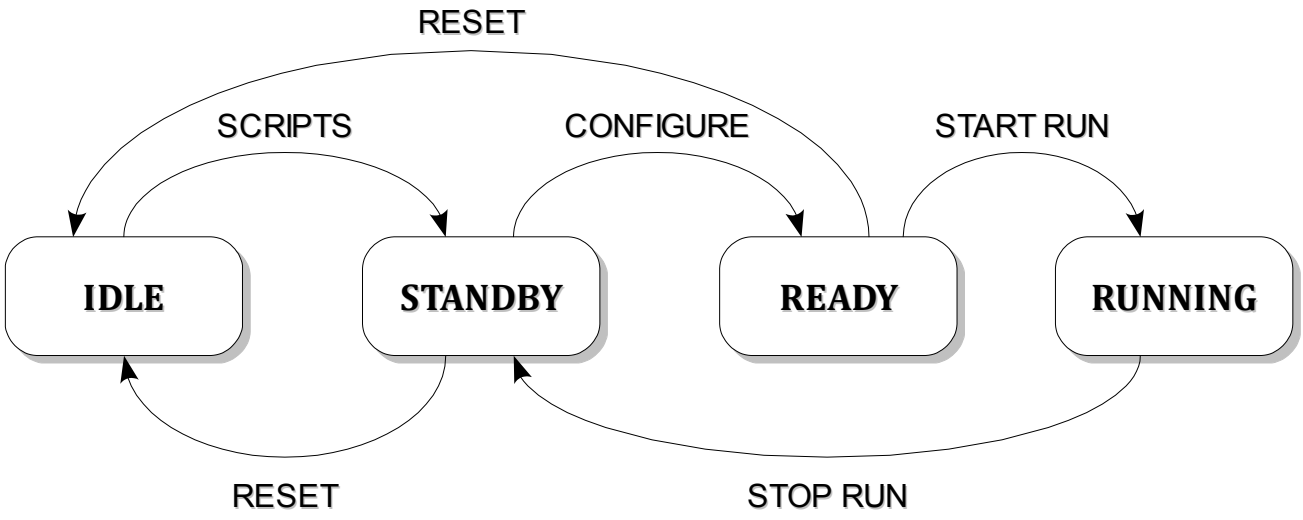


Figure 2: DCC Kit Daq State Machine

The 'Scripts' transition loads the configuration specified in the 'tree view' panel and produces all the scripts necessary to initialize and run the Daq hardware. The 'Configure' transition initializes the

hardware and the 'start run' transition starts the actual data taking.

WARNING: any modification in the configuration (on the 'tree view panel') will be actually taken into account only in transition 'Scripts'. For example, if you modify the configuration when in a state other than 'Idle' and you want that modification taken into account for the next run, you have first to issue 'Reset' and then 'Scripts'.

On Figure 1, the acquisition control panel is the one to the upper right. The topmost "sunken" label displays the current state of the acquisition; the enabled buttons represent the 'events' that the user can issue to the Daq state machine.

The Run Monitor Panel

When in state 'Running', this panel gives a number of basic informations on the current run. In other states, it gives the same information on the previous run.

'Run ID' is the run number: it is used to compute the file names associated with the run. The label on the right of 'Run ID' displays the run duration in hours, minutes and seconds.

'Run UID' is the unique ID of the run and results from the exact UTC date and time when the run started. The Run UID will be written at the beginning of the data file produced by the run. It is made of exactly 20 characters.

The 'Data (MB)' label displays the size in megabytes of the data file produced by the run.

The 'Event' label displays the number of events acquired in the run.

The 'Path' label display the default file path of the run. This path is the directory where the data and configuration files of the run are created. A click on the 'Path...' button allows the user to choose another path.

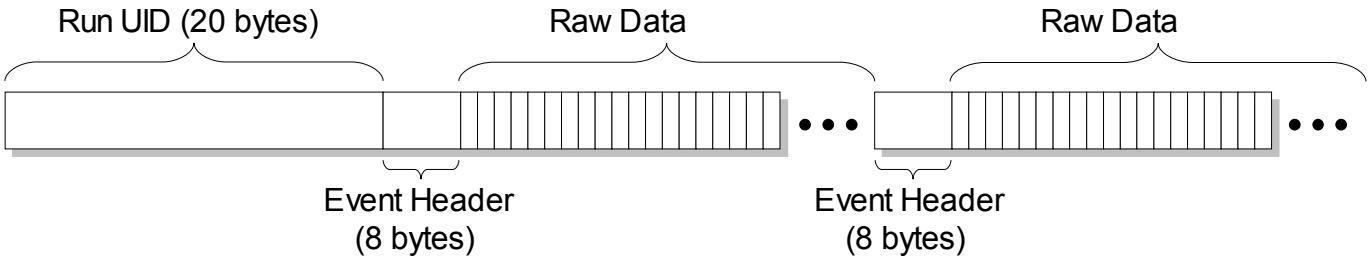
The 'Data file' and 'Config file' labels display the file names in which the data and configuration associated with the run are stored.

3. THE DATA FILE FORMAT

The data acquired by a run are stored in the file named "`RUN_XXXXX.acq`" where '`XXXXX`' is the five character long run number (with leading zeros if necessary). The file will be situated in the default path of the run. On Figure 1, the full data file path is `C:/anvar/Data/RUN_50029.acq`.

- The first 20 bytes of the data file will always begin with the Run UID, i.e. the 20 characters representing the T0 of the run. Example: '`R2007.08.23-10:12:47`'
- The rest of the file is filled with an indefinite number of events, each event being composed of ***one*** header and raw data. The raw data format is the one specified by Denis Calvet in the document entitled : *T2K TPC Read-out Electronics – Digital Front-end Mezzanine Card Design Notes*.
- The event header is 8 bytes long. It includes two 32-bits integers respectively representing the 'event size' in bytes and the 'event number'. The 'event size' is the ***total*** event size, including the 8 bytes of the header, which means that it is ***at least*** equal to 8.

The following drawing summarizes the data file format:



WARNING: All binary numbers are in "Big Endian format", also called the "network byte order" meaning that ***the most significant bytes come first***. On Intel platforms, the numbers are in "Little Endian" format so that when you read a Big Endian 4-byte integer from the data file, ***you have to swap the bytes***. This is done using the `ntohl()` and `ntohs()` standard functions respectively fort 32-bit and 16-bits integers (The meaning of the function names are 'network to host long' and 'network to host short'). To use these functions you have to include `<arpa/inet.h>` on Linux/Unix platforms and `<winsock2.h>` on Windows.

Here is an example in C/C++ showing how to read a data file:

```
#include <arpa/inet.h>
#include <stdio.h>

char rawData [200 * 1024];
struct Header
{
     int eventSize;
     int eventNumb;
}

int main()
{
     // Open binary file
     FILE* f = fopen("RUN_00001.acq", "rb");

     // Read Run UID characters
     char runUid[21];
     fread(runUid, 1, 20, f);
     runUid[20] = '\0'; // Null terminated C string

     // Event loop
     Header head;
     while (true)
     {
          // Read next header or quit of end of file
          if(fread(&head, sizeof(Header), 1, f) != 1)
          {
               fclose(f);
               return 0; // End of program
          }

          // Byte swap
          head.eventSize = ntohl(head.eventSize);
          head.eventNumb = ntohl(head.eventNumb);
```

```
            int rawDataSize = head.eventSize - sizeof(head);

            // Read Raw Data
            fread(rawData, 1, rawDataSize, f);

            /* ... DECODE AND PROCESS RAW DATA ... */
        }
}
```

4. ACCESSING THE CONFIGURATION PARAMETERS

Each time the 'Scripts' event is issued to the Daq state machine, the configuration is cloned and when 'start run' is issued, it is copied into the corresponding file: '`CFG_XXXXX.xml`' where '`XXXXX`' is the run number. As indicated by its extension, this is an XML file implementing the configuration semantics.

The users can access configuration parameters either by directly parsing the XML file or using the associated C++ API called '`CCfg`' (as "Compound Configuration").

A CCfg file is opened by declaring a `CCfg::Io::Document` object and loading the configuration stored in a file:

```
CCfg::Io::Document cfgDoc;
Ccfg::View::Object cfg(&cfgDoc.load("CFG_00023.xml"));
```

The parameters are accessed using the `()` operator. For instance, here is the way to access the value of `Daq.DCC.Fem[0].FEC[1].ASIC[3].gain` in your program:

```
int gain = cfg("Daq")("DCC")("Fem",0)("FEC",1)("ASIC",3)("gain");
```

NOTE 1: Indexes appear as a second parameter in the `()` operator. Indexes can be either integer or string values.

NOTE 2: You do not have to use the whole path every time: any object of the configuration can be stored in a corresponding variable. For instance if you want to access many parameters belonging to `Daq.DCC.Fem[0].FEC[1]`, you can store this object is a variable:

```
Ccfg::View::Attribute fec = cfg("Daq")("DCC")("Fem",0)("FEC",1);
int gain = fec("ASIC",3)("gain");
double s = fec("ASIC",3)("channel",1)("transferFunctionSlope");
```

More information (and a tutorial) on the CCfg semantics and usage can be found at the address:

`http://svom.extra.cea.fr/doc/Config/CompoundConfig/html/`