



Universiteit Gent
Faculteit Wetenschappen
Vakgroep Fysica en Sterrenkunde

2 **GIF++ DAQ**

3 An extensive documentation

4 **Alexis Fagot**

5



Data acquisition system v4.0
for CAEN VME modules v1718 &
v1190a

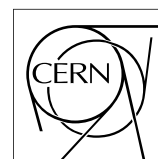


Table of Contents

7	A A data acquisition software for CAEN VME TDCs	A-1
8	A.1 GIF++ DAQ file tree	A-1
9	A.2 Usage of the DAQ	A-2
10	A.3 Description of the readout setup	A-3
11	A.4 Data read-out	A-3
12	A.4.1 V1190A TDCs	A-4
13	A.4.2 DataReader	A-6
14	A.4.3 Data quality flag	A-10
15	A.5 Communications	A-12
16	A.5.1 V1718 USB Bridge	A-13
17	A.5.2 Configuration file	A-13
18	A.5.3 WebDCS/DAQ intercommunication	A-17
19	A.5.4 Example of inter-process communication cycle	A-18
20	A.6 Software export	A-18



A data acquisition software for CAEN VME TDCs

Certifying detectors in the perspective of HL-LHC required to develop tools for the GIF++ experiment. Among them was the C++ **DAQ!** (DAQ) software that allows to make the communications in between a computer and TDC modules in order to retrieve the RPC data [1]. In this appendix, details about this software, as of how the software was written, how it functions and how it can be exported to another similar setup, will be given.

A.1 GIF++ DAQ file tree

GIF++ DAQ source code is fully available on github at https://github.com/afagot/GIF_DAQ. The software requires 3 non-optional dependencies:

- CAEN USB Driver, to mount the VME hardware,
- CAEN VME Library, to communicate with the VME hardware, and
- ROOT, to organize the collected data into a TTree.

The CAEN VME library will not be packaged by distributions and will need to be installed manually. To compile the GIF++ DAQ project via a terminal, from the DAQ folder use the command:

```
make
```

The source code tree is provided below along with comments to give an overview of the files' content. The different objects created for this project (v1718, v1190a, IniFile & DataReader) will be described in details in the following sections.

GIF_DAQ	
├── bin	
│ ├── daq.....	EXECUTABLE
│ └── include	LIST OF C++ HEADER FILES
│ ├── CAENVMElib.h.....	CAEN C++ LIBS
│ ├── CAENVMEoslib.h.....	CAEN OS C++ LIBS
│ ├── CAENVMEtypes.h	CAEN VARIABLES
│ ├── DataReader.h.....	DECLARATION OF OBJECT DATA READER
│ ├── IniFile.h.....	DECLARATION OF OBJECT INI FILE FOR INI PARSER
│ ├── MsgSvc.h.....	DECLARATION OF DAQ LOG MESSAGES
│ ├── utils.h.....	DECLARATION OF USEFUL VARIABLES AND COMM. FUNCTIONS
│ ├── v1190a.h.....	DECLARATION OF OBJECT V1190A
│ └── v1718.h.....	DECLARATION OF OBJECT V1718
│ └── lib.....	CAEN LIBRARY
│ ├── install	
│ └── x86	
│ └── libCAENVME.so.2.41	
├── obj.....	BINARY FILES CREATED BY COMPILER
│ └── ...	
├── src.....	LIST OF C++ SOURCE FILES
│ ├── daq.cxx.....	MAIN FILE
│ ├── DataReader.cxx.....	DEFINITION OF DATA READER'S METHODS
│ ├── IniFile.cxx.....	DEFINITION OF INI FILE'S METHODS
│ ├── MsgSvc.cxx.....	DEFINITION OF LOG MESSAGING FUNCTIONS
│ ├── utils.cxx.....	DEFINITION OF USEFUL COMMUNICATION FUNCTIONS
│ ├── v1190a.cxx.....	DECLARATION OF V1190A'S METHODS
│ └── v1718.cxx.....	DECLARATION OF V1718'S METHODS
├── makefile.....	COMPILER INSTRUCTIONS
└── README.md.....	README FILE FOR GITHUB

43 A.2 Usage of the DAQ

44 GIF++ DAQ, as used in GIF++, is not a standalone software. Indeed, the system being more com-
 45 plexe, the DAQ only is a sub-layer of the software architecture developped to control and monitor
 46 the RPCs that are placed into the bunker for performance study in an irradiated environment. The
 47 top layer of GIF++ is a **webDCS!** (webDCS) application. The DAQ is only called by the webDCS
 48 when data needs to be acquired. The webDCS operates the DAQ through command line. To start
 49 the DAQ, the webDCS calls:

50

```
51 bin/daq /path/to/the/log/file/in/the/output/data/folder
```

52 where `/path/to/the/log/file/in/the/output/data/folder` is the only argument required. This
 53 log file is important for the webDCS as this file contains all the content of the communication of the
 54 webDCS and the different systems monitored by the webDCS. Its content is constantly displayed
 55 during data taking for the users to be able to follow the operations. The communication messages
 56 are normally sent to the webDCS log file via the functions declared in file `MsgSvc.h`, typically
 57 `MSG_INFO(string message)`.

58

59 A.3 Description of the readout setup

60 The CMS RPC setup at GIF++ counts 5 V1190A TDC! (TDC) manufactured by CAEN [2]. V1190A
 61 are VME units accepting 128 independent Multi-Hit/Multi-Event TDC channels whose signals are
 62 treated by 4 100 ps high performance TDC chips developed by CERN / ECP-MIC Division. The
 63 communication between the computer and the TDCs to transfer data is done via a V1718 VME mas-
 64 ter module also manufactured by CAEN and operated from a USB port [3]. These VME modules
 65 are all hosted into a 6U VME 6021 powered crate manufactured by W-Ie-Ne-R than can accomodate
 66 up to 21 VME bus cards [4]. These 3 components of the DAQ setup are shown in Figure A.1.
 67

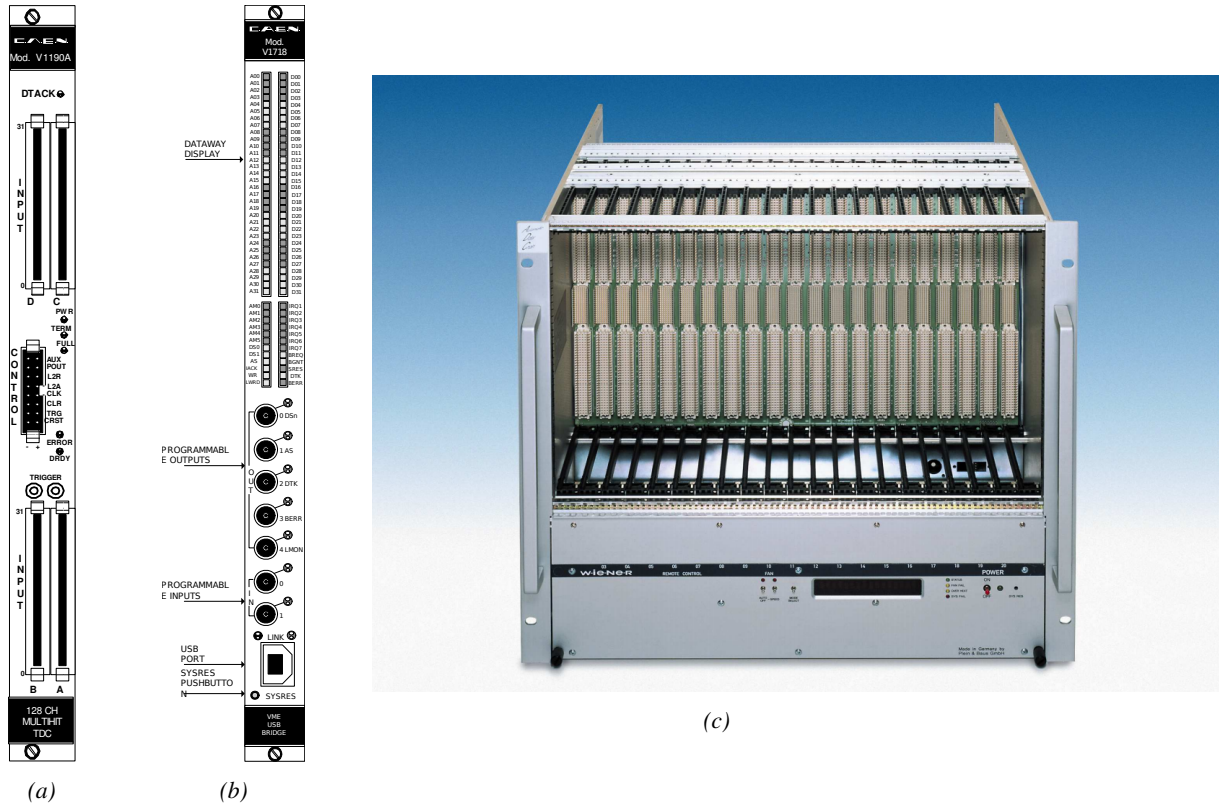


Figure A.1: (A.1a) View of the front panel of a V1190A TDC module [2]. (A.1b) View of the front panel of a V1718 Bridge module [3]. (A.1c) View of the front panel of a 6U 6021 VME crate [4].

68 A.4 Data read-out

69 To efficiently perform a data readout algorithm, C++ objects to handle the VME modules (TDCs
 70 and VME bridge) have been created along with objects to store data and read the configuration file
 71 that comes as an input of the DAQ software.
 72

A.4.1 V1190A TDCs

The DAQ used at GIF takes profit of the *Trigger Matching Mode* offered by V1190A modules. This setting is enabled through the method `v1190a::SetTrigMatching (int ntdcs)` where `ntdcs` is the total number of TDCs in the setup this setting needs to be enabled for (Source Code A.1). A trigger matching is performed in between a trigger time tag, a trigger signal sent into the TRIGGER input of the TDC visible on Figure A.1a, and the channel time measurements, signals recorded from the detectors under test in our case. Control over this data acquisition mode, explained through Figure A.2, is offered via 4 programmable parameters:

- **match window:** the matching between a trigger and a hit is done within a programmable time window. This is set via the method

```
void v1190a::SetTrigWindowWidth(Uint windowWidth, int ntdcs)
```

- **window offset:** temporal distance between the trigger tag and the start of the trigger matching window. This is set via the method

```
void v1190a::SetTrigWindowWidth(Uint windowWidth, int ntdcs)
```

- **extra search margin:** an extended time window is used to ensure that all matching hits are found. This is set via the method

```
void v1190a::SetTrigSearchMargin(Uint searchMargin, int ntdcs)
```

- **reject margin:** older hits are automatically rejected to preven buffer overflows and to speed up the search time. This is set via the method

```
void v1190a::SetTrigRejectionMargin(Uint rejectMargin, int ntdcs)
```

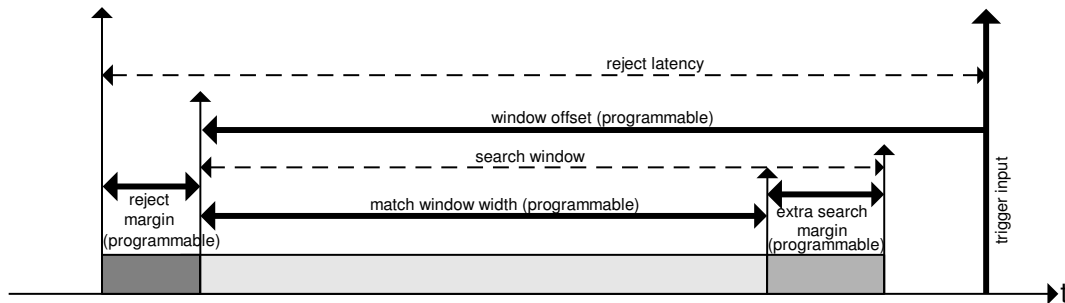


Figure A.2: Module V1190A Trigger Matching Mode timing diagram [2].

Each of these 4 parameters are given in number of clocks, 1 clock being 25 ns long. It is easy to understand at this level that there are 3 possible functioning settings:

- **1:** the match window is entirely contained after the trigger signal,
- **2:** the match window overlaps the trigger signal, or
- **3:** the match window is entirely contained before the trigger signal as displayed on Figure A.2.

In both the first and second cases, the sum of the window width and of the offset can be set to a maximum of 40 clocks, which corresponds to 1 μ s. Evidently, the offset can be negative, allowing

100 for a longer match window, with the constraint of having the window ending at most $1\text{ }\mu\text{s}$ after the
101 trigger signal. In the third case, the maximum negative offset allowed is of 2048 clocks (12 bit) cor-
102 responding to $51.2\text{ }\mu\text{s}$, the match window being strictly smaller than the offset. In the case of GIF++,
103 the choice has been made to use this last setting by delaying the trigger signal. During the studies
104 performed in GIF++, both the efficiency of the RPCs, probed using a muon beam, and the noise or
105 gamma background rate are monitored. The extra search and reject margins are left unused.

106 To probe the efficiency of RPC detectors, the trigger time tag is provided by the coincidence of
107 scintillators when a bunch of muons passes through GIF++ area is used to trigger the data acquisi-
108 tion. For this measurement, it is useful to reduce the match window width only to contain the muon
109 information. Indeed, the delay in between a trigger signal and the detection of the corresponding
110 muon in the RPC being very constant (typically a few tens of ns due to jitter and cable length), the
111 muon signals are very localised in time. Thus, due to a delay of approximately 325 ns in between
112 the muons and the trigger, the settings were chosen to have a window width of 24 clocks (600 ns)
113 centered on the muon peak thanks to a negative offset of 29 clocks (725 ns).

114 On the otherhand, monitoring the rates don't require for the DAQ to look at a specific time window.
115 It is important to integrate enough time to have a robust measurement of the rate as the number of
116 hits per time unit. The triggering signal is provided by a pulse generator at a frequency of 300 Hz
117 to ensure that the data taking occurs in a random way, with respect to beam physics, to probe only
118 the irradiation spectrum on the detectors. The match window is set to 400 clocks ($10\text{ }\mu\text{s}$) and the
119 negative offset to 401 clocks as it needs to exceed the value of the match window.

```

class v1190a
{
private :
    long          Handle;
    vector<Data32> Address;
    CVDataWidth   DataWidth;
    CVAddressModifier AddressModifier;

public:

    v1190a(long handle, IniFile *inifile, int ntdcs);
    ~v1190a();
    Data16 write_op_reg(Data32 address, int code, string error);
    Data16 read_op_reg(Data32 address, string error);
    void Reset(int ntdcs);
    void Clear(int ntdcs);
    void TestWR(Data16 value, int ntdcs);
    void CheckTDCStatus(int ntdcs);
    void CheckCommunication(int ntdcs);
    void SetTDCTestMode(Data16 mode, int ntdcs);
    void SetTrigMatching(int ntdcs);
    void SetTrigTimeSubstraction(Data16 mode, int ntdcs);
    void SetTrigWindowWidth(UINT windowWidth, int ntdcs);
    void SetTrigWindowOffset(UINT windowOffset, int ntdcs);
    void SetTrigSearchMargin(UINT searchMargin, int ntdcs);
    void SetTrigRejectionMargin(UINT rejectMargin, int ntdcs);
    void GetTrigConfiguration(int ntdcs);
    void SetTrigConfiguration(IniFile *inifile, int ntdcs);
    void SetTDCDetectionMode(Data16 mode, int ntdcs);
    void SetTDCResolution(Data16 lsb, int ntdcs);
    void SetTDCDeadTime(Data16 time, int ntdcs);
    void SetTDCHeadTrailer(Data16 mode, int ntdcs);
    void SetTDCEventSize(Data16 size, int ntdcs);
    void SwitchChannels(IniFile *inifile, int ntdcs);
    void SetIRQ(Data32 level, Data32 count, int ntdcs);
    void SetBlockTransferMode(Data16 mode, int ntdcs);
    void Set(IniFile *inifile, int ntdcs);
    void CheckStatus(CVErrorCodes status) const;
    int ReadBlockD32(UINT tdc, const Data16 address,
                    Data32 *data, const UINT words, bool ignore_berr);
    UINT Read(RAWData *DataList, int ntdcs);
};

```

Source Code A.1: Description of C++ object v1190a.

The v1190a object, defined in the DAQ software as in Source Code A.1, offers the possibility to concatenate all TDCs in the readout setup into a single object containing a list of hardware addresses (addresses to access the TDCs' buffer through the VME crate) and each constructor and method acts on the list of TDCs.

A.4.2 DataReader

Enabled thanks to `v1190a::SetBlockTransferMode(Data16 mode, int ntdcs)`, the data transfer is done via **BLT!** (BLT). Using BLT allows to transfer a fixed number of events called a *block*. This is used together with an **AFL!** (AFL) of the TDCs' output buffers, defined through `v1190a::SetIRQ(Data32`

level, Data32 count, **int** ntdcs). This AFL gives the maximum amount of 32735 words (16 bits, corresponding to the depth of a TDC output buffer) that can be written in a buffer before an **IRQ!** (IRQ) is generated and seen by the VME Bridge, stopping the data acquisition to transfer the content of each TDC buffers before resuming. For each trigger, 6 words or more are written into the TDC buffer:

- **a global header** providing information of the event number since the beginning of the data acquisition,
- **a TDC header,**
- **the TDC data** (*if any*), 1 for each hit recorded during the event, providing the channel and the time stamp associated to the hit,
- **a TDC error** providing error flags,
- **a TDC trailer,**
- **a global trigger time tag** that provides the absolute trigger time relatively to the last reset, and
- **a global trailer** providing the total word count in the event.

As previously described in Section ??, CMS RPC FEEs provide us with 100 ns long LVDS output signals that are injected into the TDCs' input. Any avalanche signal that gives a signal above the FEEs threshold is thus recorded by the TDCs as a hit within the match window. Each hit is assigned to a specific TDC channel with a time stamp, with a precision of 100 ps. The reference time, $t_0 = 0$, is provided by the beginning of the match window. Thus for each trigger, coming from a scintillator coincidence or the pulse generator, a list of hits is stored into the TDCs' buffers and will then be transferred into a ROOT Tree.

When the BLT is used, it is easy to understand that the maximum number of words that have been set as ALF will not be a finite number of events or, at least, the number of events that would be recorded into the TDC buffers will not be a multiple of the block size. In the last BLT cycle to transfer data, the number of events to transfer will most probably be lower than the block size. In that case, the TDC can add fillers at the end of the block but this option requires to send more data to the computer and is thus a little slower. Another solution is to finish the transfer after the last event by sending a bus error that states that the BLT reached the last event in the pile. This method has been chosen in GIF++.

Due to irradiation, an event in GIF++ can count up to 300 words per TDC. A limit of 4096 words (12 bits) has been set to generate IRQ which represent from 14 to almost 700 events depending on the average of hits collected per event. Then the block size has been set to 100 events with enabled bus errors. When an AFL is reached for one of the TDCs, the VME bridge stops the acquisition by sending a BUSY signal.

The data is then transfered one TDC at a time into a structure called `RAWData` (Source Code A.2).

```

struct RAWData{
    vector<int>          *EventList;
    vector<int>          *NHitsList;
    vector<int>          *QFlagList;
    vector<vector<int> > *ChannelList;
    vector<vector<float> > *TimeStampList;
};

```

Source Code A.2: Description of data holding C++ structure `RAWData`.

In order to organize the data transfer and the data storage, an object called `DataReader` was created (Source Code A.3). On one hand, it has `v1718` and `v1190a` objects as private members for communication purposes, such as VME modules settings via the configuration file `*iniFile` or data read-out through `v1190a::Read()` and on the other hand, it contains the structure `RAWData` that allows to organise the data in vectors reproducing the tree structure of a ROOT file.

```

class DataReader
{
private:
    bool        StopFlag;
    IniFile     *iniFile;
    Data32      MaxTriggers;
    v1718       *VME;
    int         nTDCs;
    v1190a      *TDCs;
    RAWData     TDCData;

public:
    DataReader();
    virtual ~DataReader();
    void        SetIniFile(string inifilename);
    void        SetMaxTriggers();
    Data32      GetMaxTriggers();
    void        SetVME();
    void        SetTDC();
    int         GetQFlag(UINT it);
    void        Init(string inifilename);
    void        FlushBuffer();
    void        Update();
    string      GetFileName();
    void        WriteRunRegistry(string filename);
    void        Run();
};

```

Source Code A.3: Description of C++ object `DataReader`.

Each event is transferred from `TDCData` and saved into branches of a ROOT `TTree` as 3 integers that represent the event ID (`EventCount`), the number of hits read from the TDCs (`nHits`), and the quality flag that provides information for any problem in the data transfer (`qflag`), and 2 lists of `nHits` elements containing the fired TDC channels (`TDCh`) and their respective time stamps (`TDCTS`), as presented in Source Code A.4. The ROOT file is named using information contained into the configuration file, presented in section A.5.2. The needed information is extracted using method `DataReader::GetFileName()` and allow to build the output filename format `ScanXXXXXX_HVX_DAQ.root`

189 where ScanXXXXXX is a 6 digit number representing the scan number into GIF++ database and HVX
 190 the HV step within the scan that can be more than a single digit. An example of ROOT data file is
 191 provided with Figure A.3.

```

192 RAWData TDCData;
    TFile *outputFile = new TFile(outputFileName.c_str(), "recreate");
    TTree *RAWDataTree = new TTree("RAWData", "RAWData");

    int          EventCount = -9;
    int          nHits = -8;
    int          qflag = -7;
    vector<int>   TDCCh;
    vector<float> TDCTS;

    RAWDataTree->Branch("EventNumber", &EventCount, "EventNumber/I");
    RAWDataTree->Branch("number_of_hits", &nHits, "number_of_hits/I");
    RAWDataTree->Branch("Quality_flag", &qflag, "Quality_flag/I");
    RAWDataTree->Branch("TDC_channel", &TDCCh);
    RAWDataTree->Branch("TDC_TimeStamp", &TDCTS);

193 //...
    //Here read the TDC data using v1190a::Read() and place it into
    //TDCData for as long as you didn't collect the requested amount
    //of data.
    //...

    for( Uint i=0; i<TDCData.EventList->size(); i++){
        EventCount = TDCData.EventList->at(i);
        nHits      = TDCData.NHitsList->at(i);
        qflag      = TDCData.QFlagList->at(i);
        TDCCh      = TDCData.ChannelList->at(i);
        TDCTS      = TDCData.TimeStampList->at(i);
        RAWDataTree->Fill();
    }
  
```

194 *Source Code A.4: Highlight of the data transfer and organisation within DataReader::Run() after the data has been collected into TDCData.*

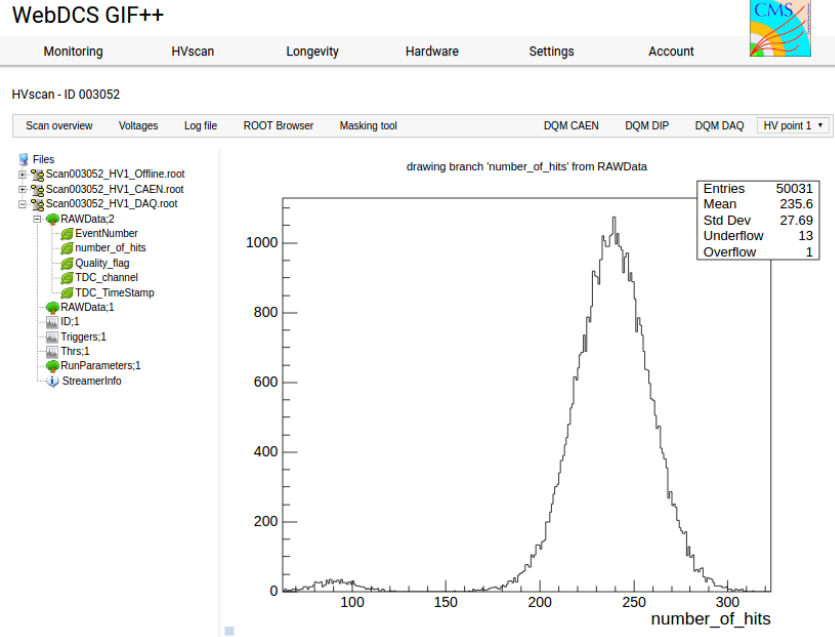


Figure A.3: Structure of the ROOT output file generated by the DAQ. The 5 branches (*EventNumber*, *number_of_hits*, *Quality_flag*, *TDC_channel* and *TDC_TimeStamp*) are visible on the left panel of the ROOT browser. On the right panel is visible the histogram corresponding to the variable *nHits*. In this specific example, there were approximately 50k events recorded to measure the gamma irradiation rate on the detectors. Each event is stored as a single entry in the *TTree*.

A.4.3 Data quality flag

Among the parameters that are recorded for each event, the quality flag, defined in Source Code A.5, is determined on the fly by checking the data recorded by every single TDC. From method `v1190a::Read()`, it can be understood that the content of each TDC buffer is readout one TDC at a time. Entries are created in the data list for the first TDC and then, when the second buffer is readout, events corresponding to entries that have already been created to store data for the previous TDC are added to the existing list element. On the contrary, when an event entry has not been yet created in the data list, a new entry is created.

```
typedef enum _QualityFlag {
    GOOD      = 1,
    CORRUPTED = 0
} QualityFlag;
```

Source Code A.5: Definition of the quality flag `enum`.

It is possible that each TDC buffer contains a different number of events. In cases where the first element in the buffer list is an event for corresponds to a new entry, the difference in between the entry from the buffer and the last entry in the data list is recorded and checked. If it is greater than 1, what should never be the case, the quality flag is set to `CORRUPTED` for this TDC and an empty entry is created in the place of the missing ones. Missing entries are believe to be the result of a bad hold

on the TDC buffers at the moment of the readout. Indeed, the software hold is effective only on 1 TDC at a time and no solution as been found yet to completely block the writting in the buffers when an IRQ is received.

At the end of each BLT cycle, the ID of the last entry stored for each TDC buffer is not recorded. When starting the next cycle, if the first entry in the pile corresponds to an event already existing in the list, the readout will start from this list element and will not be able to check the difference in between this entry's ID and the one of the last entry that was recorded for this TDC buffer in the previous cycle. In the case events were missing, the flag stays at its initial value of 0, which is similar to `CORRUPTED` and it is assumed that then this TDC will not contribute to `number_of_hits`, `TDC_channel` or `TDC_TimeStamp`.

Finally, since there will be 1 `RAWData` entry per TDC for each event (meaning `nTDCs` entries, referring to `DataReader` private attribute), the individual flags of each TDC will be added together. The final format is an integer composed `nTDCs` digits where each digit is the flag of a specific TDC. This is constructed using powers of 10 like follows:

$$\text{TDC 0: } QFlag = 10^0 \times _QualityFlag$$

$$\text{TDC 1: } QFlag = 10^1 \times _QualityFlag$$

...

$$\text{TDC N: } QFlag = 10^N \times _QualityFlag$$

and the final flag to be with N digits:

$$QFlag = n \dots 3210$$

each digit being 1 or 0. Bellow is given an example with a 4 TDCs setup.

$$\text{If all TDCs were good : } QFlag = 1111,$$

$$\text{but if TDC 2 was corrupted : } QFlag = 1011.$$

When data taking is over and the data contained in the dynamical `RAWData` structure is transfered to the ROOT file, all the 0s are changed into 2s by calling the method `DataReader::GetQFlag()`. This will help translating the flag without knowing the number of TDCs beforehand. Indeed, a flag 111 could be due to a 3 TDC setup with 3 good individual TDC flags or to a more than 3 TDC setup with TDCs those ID is greater than 2 being `CORRUPTED`, thus giving a 0.

The quality flag has been introduced quite late, in October 2017 only, to the list of GIF++ DAQ parameters to be recorded into the output ROOT file. Before this addition, the missing data, corrupting the quality for the offline analysis, was contributing to artificially fill data with lower multiplicity. Looking at `TBranch number_of_hits` provides an information about the data of the full GIF++ setup. When a TDC is not able to transfer data for a specific event, the effect is a reduction of the total number of hits recorded in the full setup, this is what can be seen from Figure A.4. After offline reconstruction detector by detector, the effect of missing events can be seen in the artificially filled bin at multiplicity 0 shown in Figure A.5. Nonetheless, for data with high irradiation levels, as it is the case for Figure A.5a, discarding the fake multiplicity 0 data can be done easily during the offline analysis. At lower radiation, the missing events contribution becomes more problematic as the multiplicity distribution overlaps the multiplicity 0 and that in the same time the proportion of missing

events decreases. Attempts to fit the distribution with a Poisson or skew distribution function were not conclusive and this very problem has been at the origin of the quality flag that allows to give a non ambiguous information about each event quality.

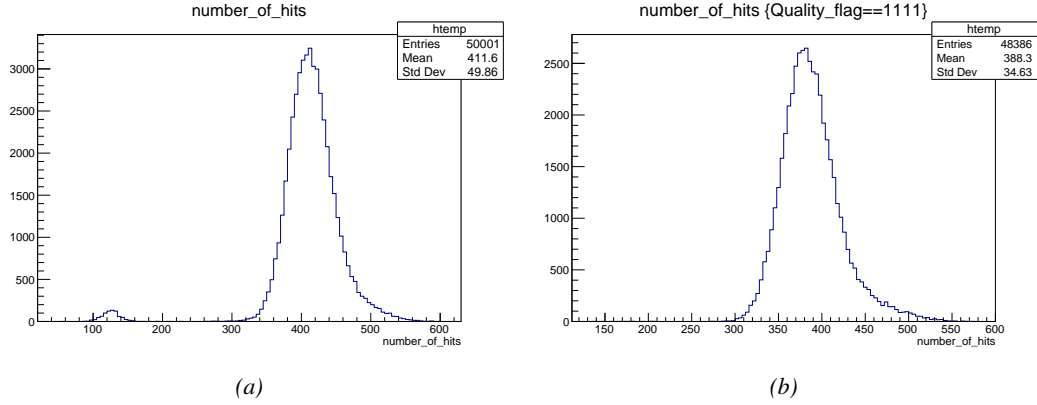


Figure A.4: The effect of the quality flag is explained by presenting the content of TBranch `number_of_hits` of a data file without `Quality_flag` in Figure A.4a and the content of the same TBranch for data corresponding to a `Quality_flag` where all TDCs were labelled as `GOOD` in Figure A.4b taken with similar conditions. It can be noted that the number of entries in Figure A.4b is slightly lower then in Figure A.4a due to the excluded events.

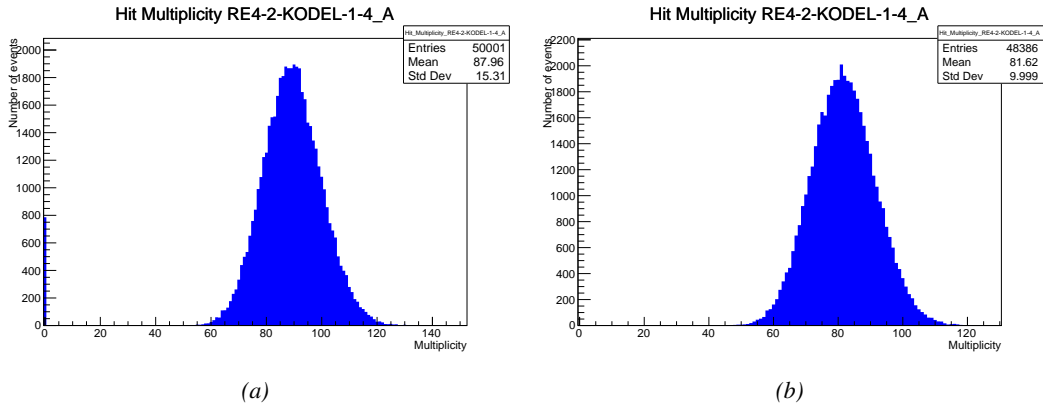


Figure A.5: Using the same data as previously showed in Figure A.4, the effect of the quality flag is explained by presenting the reconstructed hit multiplicity of a data file without `Quality_flag` in Figure A.5a and the reconstructed content of the same RPC partition for data corresponding to a `Quality_flag` where all TDCs were labelled as `GOOD` in Figure A.5b taken with similar conditions. The artificial high content of bin 0 is completely suppressed.

A.5 Communications

To ensure data readout and dialog in between the machine and the TDCs or in between the webDCS and the DAQ, different communication solutions were used. First of all, it is important to have a

module to allow the communication in between the TDCs and the computer from which the DAQ operates. When this communication is effective, shifters using the webDCS to control data taking can thus send instructions to the DAQ.

A.5.1 V1718 USB Bridge

In the previous section, the data transfer has been discussed. The importance of the `v1718` object (Source Code A.6), used as private member of `DataReader`, was not explicated. VME master modules are used for communication purposes as they host the USB port that connects the powered crate buffer to the computer where the DAQ is installed. From the source code point of view, this object is used to control the communication status, by reading the returned error codes with `v1718::CheckStatus()`, or to check for IRQs coming from the TDCs through `v1718::CheckIRQ()`. Finally, to ensure that triggers are blocked at the hardware level, a NIM pulse is sent out of one of the 5 programmable outputs (`v1718::SendBUSY()`) to the VETO of the coincidence module where the trigger signals originate from. As long as this signal is ON, no trigger can reach the TDCs anymore.

```
class v1718{
private:
    int          Handle;
    Data32       Data;           // Data
    CVIRQLevels  Level;          // Interrupt level
    CVAddressModifier AM;        // Addressing Mode
    CVDataWidth  DataSize;       // Data Format
    Data32       BaseAddress;    // Base Address

public:
    v1718(IniFile *inifile);
    ~v1718();
    long          GetHandle(void) const;
    int           SetData(Data16 data);
    Data16        GetData(void);
    int           SetLevel(CVIRQLevels level);
    CVIRQLevels   GetLevel(void);
    int           SetAM(CVAddressModifier am);
    CVAddressModifier GetAM(void);
    int           SetDataSize(CVDataWidth datasize);
    CVDataWidth   GetDataSize(void);
    int           SetBaseAddress(Data16 baseaddress);
    Data16        GetBaseAddress(void);
    void          CheckStatus(CVErrorCodes status) const;
    bool          CheckIRQ();
    void          SetPulsers();
    void          SendBUSY(BusyLevel level);
};
```

Source Code A.6: Description of C++ object `v1718`.

A.5.2 Configuration file

The DAQ software takes as input a configuration file written using INI standard [5]. This file is partly filled with the information provided by the shifters when starting data acquisition using the webDCS, as shown by Figure A.6. This information is written in section `[General]` and will later

be stored in the ROOT file that contains the DAQ data as can be seen from Figure A.3. Indeed, another TTree called `RunParameters` as well as the 2 histograms `ID`, containing the scan number, start and stop time stamps, and `Triggers`, containing the number of triggers requested by the shifter, are available in the data files. Moreover, `ScanID` and `HV` are then used to construct the file name thanks to the method `DataReader::GetFileName()`.

WebDCS GIF++

Monitoring HVscan Longevity Hardware Settings Account

DAQ High Voltage Scan

Type scan:

Source configuration: U 333 D 333

Beam configuration:

Waiting time: (min)

Trigger mode: ☐ External ☐ Internal ☐ Random

Minimal measure time: (min)

Comments:

HV after scan:

Chamber	RE2-2-NPD-BARC-8	RE4-2-CERN-106	RE3-2-NPD-BARC-9	RE4-2-CERN-105	RE4-2-KODEL-1-4	Max triggers
HV _{ref} 1	8600	8500	8600	8500	6500	<input type="text"/>
HV _{ref} 2	8700	8600	8700	8600	6600	<input type="text"/>
HV _{ref} 3	8800	8700	8800	8700	6700	<input type="text"/>
HV _{ref} 4	8900	8800	8900	8800	6800	<input type="text"/>
HV _{ref} 5	9000	8900	9000	8900	6900	<input type="text"/>
HV _{ref} 6	9100	9000	9100	9000	7000	<input type="text"/>
HV _{ref} 7	9200	9100	9200	9100	7100	<input type="text"/>
HV _{ref} 8	9300	9200	9300	9200	7200	<input type="text"/>
HV _{ref} 9	9400	9300	9400	9300	7300	<input type="text"/>
HV _{ref} 10	9500	9400	9500	9400	7400	<input type="text"/>

Figure A.6: WebDCS DAQ scan page. On this page, shifters need to choose the type of scan (Rate, Efficiency or Noise Reference scan), the gamma source configuration at the moment of data taking, the beam configuration, and the trigger mode. These information will be stored in the DAQ ROOT output. Are also given the minimal measurement time and waiting time after ramping up of the detectors is over before starting the data acquisition. Then, the list of HV points to scan and the number of triggers for each run of the scan are given in the table underneath.

The rest of the information is written beforehand in the configuration file template, as explicated in Source Code A.7, and contains the hardware addresses to the different VME modules in the setup as well as settings for the TDCs. As the TDC settings available in the configuration file are not supposed to be modified, an improvement would be to remove them from the configuration file and to hardcode them inside of the DAQ code itself or to place them into a different INI file that would host only the TDC settings to lower the probability for a bad manipulation of the configuration file that can be modified from one of webDCS' menus.

290

```
[General]
TdcS=4
ScanID=$scanid
HV=$HV
RunType=$runtype
MaxTriggers=$maxtriggers
Beam=$beam
```

[VMEInterface]

```
Type=V1718
BaseAddress=0xFF0000
Name=VmeInterface
```

[TDC0]

```
Type=V1190A
BaseAddress=0x00000000
Name=Tdc0
StatusA00-15=1
StatusA16-31=1
StatusB00-15=1
StatusB16-31=1
StatusC00-15=1
StatusC16-31=1
StatusD00-15=1
StatusD16-31=1
```

[TDC1]

```
Type=V1190A
BaseAddress=0x11110000
Name=Tdc1
StatusA00-15=1
StatusA16-31=1
StatusB00-15=1
StatusB16-31=1
StatusC00-15=1
StatusC16-31=1
StatusD00-15=1
StatusD16-31=1
```

291

[TDC2]

```
Type=V1190A
BaseAddress=0x22220000
Name=Tdc2
StatusA00-15=1
StatusA16-31=1
StatusB00-15=1
StatusB16-31=1
StatusC00-15=1
StatusC16-31=1
StatusD00-15=1
StatusD16-31=1
```

[TDC3]

```
Type=V1190A
BaseAddress=0x44440000
Name=Tdc3
StatusA00-15=1
StatusA16-31=1
StatusB00-15=1
StatusB16-31=1
StatusC00-15=1
StatusC16-31=1
StatusD00-15=1
StatusD16-31=1
```

[TDCSettings]

```
TriggerExtraSearchMargin=0
TriggerRejectMargin=0
TriggerTimeSubtraction=0b1
TdcDetectionMode=0b01
TdcResolution=0b10
TdcDeadTime=0b00
TdcHeadTrailer=0b1
TdcEventSize=0b1001
TdcTestMode=0b0
BLTMode=1
```

Source Code A.7: INI configuration file template for 4 TDCs. In section **[General]**, the number of TDCs is explicated and information about the ongoing run is given. Then, there are sections for each and every VME modules. There buffer addresses are given and for the TDCs, the list of channels to enable is given. Finally, in section **[TDCSettings]**, a part of the TDC settings are given.

In order to retrieve the information of the configuration file, the object `IniFile` has been developed to provide an INI parser, presented in Source Code A.8. It contains private methods returning a boolean to check the type of line written in the file, whether a comment, a group header or a key line (`IniFile::CheckIfComment()`, `IniFile::CheckIfGroup()` and `IniFile::CheckIfToken()`). The key may sometimes be referred to as *token* in the source code. Moreover, the private element `FileData` is a map of `const string` to `string` that allows to store the data contained inside the configuration file via the public method `IniFile::GetFileData()` following the formatting (see method `IniFile::Read()`):

```
string group, token, value;  
// Get the field values for the 3 strings.  
// Then concatenate group and token together as a single string  
// with a dot separation.  
token = group + "." + token;  
FileData[token] = value;
```

More methods have been written to translate the different keys into the right variable format when used by the DAQ. For example, to get a `float` value out of the configuration file data, knowing the group and the key needed, the method `IniFile::floatType()` can be used. It takes 3 arguments being the group name and key name (both `string`), and a default `float` value used as exception in the case the expected combination of group and key cannot be found in the configuration file. This default value is then used and the DAQ continues on working after sending an alert in the log file for further debugging.

```

310 typedef map< const string, string > IniFileData;

class IniFile{
private:
    bool        CheckIfComment(string line);
    bool        CheckIfGroup(string line, string& group);
    bool        CheckIfToken(string line, string& key, string& value);
    string      FileName;
    IniFileData FileData;
    int         Error;

public:
    IniFile();
    IniFile(string filename);
    virtual ~IniFile();

    // Basic file operations
    void        SetFileName(string filename);
    int         Read();
    int         Write();
    IniFileData GetFileData();

    // Data readout methods
    Data32      addressType (string groupname, string keyname, Data32
↪ defaultvalue);
    long        intType     (string groupname, string keyname, long
↪ defaultvalue);
    long long   longType    (string groupname, string keyname, long long
↪ defaultvalue );
    string      stringType  (string groupname, string keyname, string
↪ defaultvalue );
    float       floatType   (string groupname, string keyname, float
↪ defaultvalue );

    // Error methods
    string      GetErrorMsg();
};

```

Source Code A.8: Description of C++ object *IniFile* used as a parser for INI file format.

A.5.3 WebDCS/DAQ intercommunication

When shifters send instructions to the DAQ via the configuration file, it is the webDCS itself that gives the start command to the DAQ and then the 2 softwares use inter-process communication through file to synchronise themselves. This communication file is represented by the variable `const string __runstatuspath`.

On one side, the webDCS sends commands or status that are readout by the DAQ:

- INIT, status sent when launching a scan and read via function `CtrlRunStatus(...)`,
- START, command to start data taking and read via function `CheckSTART()`,
- STOP, command to stop data taking at the end of the scan and read via function `CheckSTOP()`, and
- KILL, command to kill data taking sent by user and read via function `CheckKILL()`

and on the other, the DAQ sends status that are controlled by the webDCS:

- DAQ_RDY, sent with `SendDAQReady()` to signify that the DAQ is ready to receive commands from the webDCS,
- RUNNING, sent with `SendDAQRunning()` to signify that the DAQ is taking data,
- DAQ_ERR, sent with `SendDAQError()` to signify that the DAQ didn't receive the expected command from the webDCS or that the launch command didn't have the right number of arguments,
- RD_ERR, sent when the DAQ wasn't able to read the communication file, and
- WR_ERR, sent when the DAQ wasn't able to write into the communication file.

A.5.4 Example of inter-process communication cycle

Under normal conditions, the webDCS and the DAQ processes exchange commands and status via the file hosted at the address `__runstatuspath`, as explained in subsection A.5.3. An example of cycle is given in Table A.1. In this example, the steps 3 to 5 are repeated as long as the webDCS tells the DAQ to take data. A data taking cycle is the equivalent as what is called a *Scan* in GIF++ jargon, referring to a set of runs with several HV steps. Each repetition of steps 3 to 5 is then equivalent to a single *Run*.

At any moment during the data taking, for any reason, the shifter can decide that the data taking needs to be stopped before it reached the end of the scheduled cycle. Thus at any moment on the cycle, the content of the inter-process communication file will be changed to `KILL` and the DAQ will shut down right away. The DAQ checks for `KILL` signals every 5s after the TDCs configuration is over. So far, the function `CheckKILL()` has been used only inside of the data taking loop of method `DataReader::Run()` and thus, if the shifter decides to KILL the data taking during the TDC configuration phase or the HV ramping in between 2 HV steps, the DAQ will not be stopped smoothly and a *force kill* command will be sent to stop the DAQ process that is still awake on the computer. Improvements can be brought on this part of the software to make sure that the DAQ can safely shutdown at any moment.

A.6 Software export

In section A.2 was discussed the fact that the DAQ as written in its last version is not a standalone software. It is possible to make it a standalone program that could be adapted to any VME setup using V1190A and V1718 modules by creating a GUI for the software or by printing the log messages that are normally printed in the webDCS through the log file, directly into the terminal. This method was used by the DAQ up to version 3.0 moment where the webDCS was completed. Also, it is possible to check branches of DAQ v2.X to have example of communication through a terminal.

DAQ v2.X is nonetheless limited in its possibilities and requires a lot of offline manual interventions from the users. Indeed, there is no communication of the software with the detectors' power supply system that would allow for a user to predefine a list of voltages to operate the detectors at

step	actions of webDCS	status of DAQ	__runstatuspath
1	launch DAQ ramp voltages ramping over wait for currents stabilization	readout of IniFfile configuration of TDCs	INIT
2		configuration done send DAQ ready wait for START signal	DAQ_RDY
3	waiting time over send START		START
4	wait for run to end monitor DAQ run status	data taking ongoing check for KILL signal	RUNNING
5		run over send DAQ_RDY wait for next DCS signal	DAQ_RDY
6	ramp voltages ramping over wait for currents stabilization		DAQ_RDY
3	waiting time over send START		START
4	wait for run to end monitor DAQ run status	update IniFile information data taking ongoing check for KILL signal	RUNNING
5		run over send DAQ_RDY wait for next DCS signal	DAQ_RDY
7	send command STOP	DAQ shuts down	STOP

Table A.1: Inter-process communication cycles in between the webDCS and the DAQ through file string signals.

and loop over to take data without any further manual intervention. In v2.X, the data is taken for a single detector setting and at the end of each run, the software asks the user if he intends on taking more runs. If so, the software invites the user to set the operating voltages accordingly to what is necessary and to manually update the configuration file in consequence. This working mode can be a very first approach before an evolution and has been successfully used by colleagues from different collaborations.

For a more robust operation, it is recommended to develop a GUI or a web application to interface the DAQ. Moreover, to limit the amount of manual interventions, and thus the probability to make mistakes, it is also recommended to add an extra feature into the DAQ by installing the HV Wrapper library provided by CAEN of which an example of use in a similar DAQ software developed by a master student of UGent, and called TinyDAQ, is provided on UGent's github. Then, this HV Wrapper will help you communicating with and give instructions to a CAEN HV powered crate and can be added into the DAQ at the same level where the communication with the user was made in DAQ v2.X. In case you are using another kind of power system for your detectors, it is stringly advised to use HV modules or crates that can be remotely controled via a using C++ libraries.

References

- 381 [1] A. Fagot. *GIF++ DAQ v4.0*. 2017. URL: https://github.com/afagot/GIF_DAQ.
- 382 [2] CAEN. *Mod. V1190-VX1190 A/B, 128/64 Ch Multihit TDC*. 14th ed. 2016.
- 383 [3] CAEN. *Mod. V1718 VME USB Bridge*. 9th ed. 2009.
- 384 [4] W-Ie-Ne-R. *VME 6021-23 VXI*. 5th ed. 2016.
- 385 [5] Wikipedia. *INI file*. 2017. URL: https://en.wikipedia.org/wiki/INI_file.