NA62 Utilitites, Progress & DAQling Feature Implementations, Refactors, and Bug Fixes

Viktor Vilhelm Sonesten
Luleå Technical University, Sweden
CERN Summer Student 2019
<vikson-6@student.ltu.se>

As supervised by:
Enrico Gamberini <engamber@cern.ch>
Roland Sipos <rsipos@cern.ch>

August 30, 2019

Abstract

During the Summer of 2019 (May 10–August 30) I was contracted by CERN as a Summer Student in EP-DT-DI to help upgrade the data acquisition system (DAQ) in NA62 to conform to upgraded readout system hardware. While that was my main project, I started contributing to DAQling (a DAQ framework) halfway through my contract which is what the bulk of my contributions have been made towards. This report briefly introduces the projects and extensively covers the contributions I have made during my internship.

1 NA62 and Contributions Made

NA62 is an experiment for precision tests of the Standard Model by studying rare decays. [2] The reason for the DAQ upgrade is mainly two-fold: the CERN accelerator complex is temporarily shut down which presents an opportunity for general upgrades, and because there is new time-to-digital converter (TDC) hardware in development to replace the currently used TEL62. Due to its higher data rate, we must also upgrade the hardware used downstream in the data flow to cope with the higher data throughput. For downstream, we aim to use FELIX, the detector readout sysem used in ATLAS. [1]

The planned upgrade is as follows (visualized in Fig. 1):

1. readout data from NA62 is fed via multiple channels to FELIX cards;
2. the cards communicate their data with the `felixcore` software on the system they are installed;
3. `felixcore` splits acquired data into chunks and sends these to the trigger matching software (TMS);
4. the TMS sorts received chunks unto a timeline representing the burst, and sends any requested data to the processing farm.

1.1 bin2felix

Of note in `felixcore` is the `--file` option that allows us to emulate the NA62 and FELIX components in Fig. 1 using an appropriate input file. To this end I created `bin2felix` [5], a binary file converter that takes any input file and the optional parameters:

- `-w WORD-SIZE`: the size of each word, in bytes;

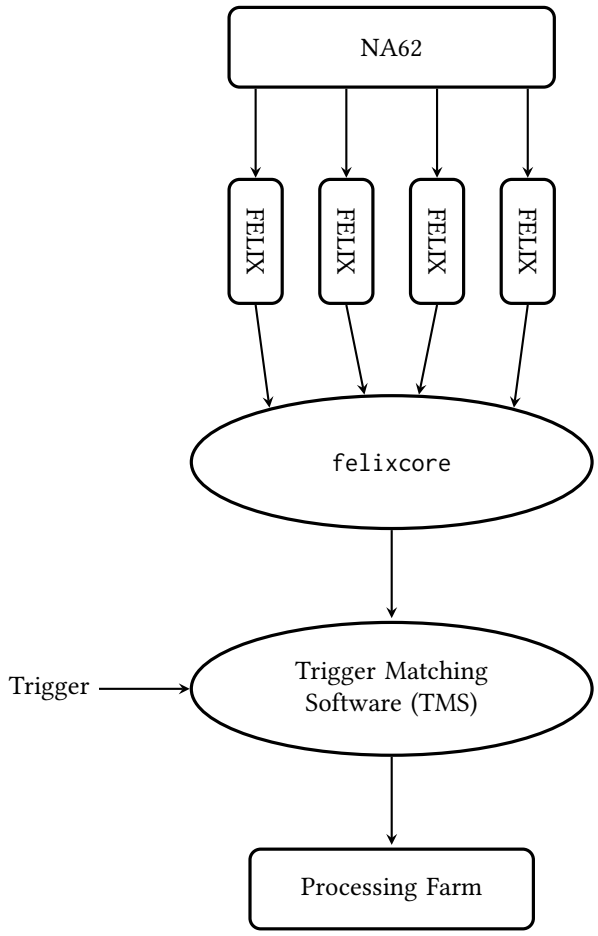

Figure 1: Visualization of the wanted NA62 DAQ upgrade.

- `-p WORDS-PER-CHUNK`: how many words each chunk should consist of;
- `-e ELINKS`: how many channels of data that should be emulated; and
- `-s WORD-TYPE`: whether input data should be sorted;

to generate a `felixcore`-compliant input file. Using this file, the dependency on upsteam hardware can be removed for any changes done to the software in the processing farm.

`bin2felix` is not a NA62-specitic utility and can ease development for any other experiment that aims to use the FELIX system in the future.

1.2 process_hits

Before NA62 test data can be converted using `bin2felix` it must first be converted from its raw form to a binary format that is easily machine-readable. This utility existed before my contract began but it has seen some performance tweaks: a generic file conversion now takes 15 seconds instead of ∼5 minutes. The reason for this performance difference is the usage of dynamic memory allocations. In the previous version the file was first read wholly into memory before any data was processed. In the new version, data is read word-for-word from the input file, processed, and written to file; only memory that is truly required is allocated, and now on the stack instead of the heap. Available at [6].

1.3 Concept Trigger Matching Software Implementation

Aside from the utilities listed above, I also wrote a concept implementation of the TMS. [3] Sorting chunks received from `felixcore` unto a timeline representing the burst as wanted, it listens to requests from an input file and counts the number of words read out. Work has also begun to listen to similar requests from UDP packets and respond with an appropriate data package.

Problems with a NA62-internal UDP-packet sender halted further development for the time being. It was decided I would focus the remainder of my internship on DAQling.

2 DAQling and Contributions Made

DAQling is a framework written in C++ for generic data acquisition. Its aim is to be as general as possible; if you are in need of data acquisition in some form you can use DAQling as a base to implement whatever modules you require (it they are not implemented already). Given a description of the data flow in JSON, configured modules are loaded during runtime. [4]

It is currently employed by FASER and under evaluation by other experiments/projects.

2.1 FileWriterModule

Starting from a concept implementation the FileWriterModule (the module responsible for writing your acquired data to file) is now production ready. The module takes the following configuration parameters:

- `max_filesize`: maximum filesize before output file rotates. Payloads received from other modules are atomic and are never split over multiple output files.
- `buffer_size`: how much data that should be buffered in memory before it is flushed to file.
- `filename_pattern`: a prinf-like format string that expands into a filename during runtime. E.g., `/tmp/output-%D.%c.%n` can expand into `/tmp/output-2019-08-16-17:14:55.1.0.bin` depending on context. `%D` expands into the current date and timestamp on the form `YYYY-MM-DD-HH:MM:SS`; `%c` into the channel ID; and `%n` into the current file number (how many times files have been rotated).

The module also registers the following variables for metrics using DAQling's statistics subsystem:

- `DL_BytesWritten_chid<N>`: number of bytes flushed to file;
- `DL_PayloadQueueSize_chid<N>`: how many payloads currently stored in the buffer;
- `DL_PayloadQueueBytes_chid<N>`: number of total bytes currently stored in the buffer;

where <N> is the channel ID.

Relevant merge requests: !17, !29, !35, !40, !45.

2.2 Revamped CMake Configuration

A big goal of DAQling is to remove as much module author overhead as possible so that new developers may focus on their module implementations. To this end the CMake configuration has been revamped. Previously, one would add a module in this manner:

```
// Before (source file)
extern "C" SomeModule *daqling_create_module() {
  return new SomeModule();
}
extern "C" void daqling_destroy_module(SomeModule
↪  *object) {
    delete object;
}

SomeModule::SomeModule() {}
SomeModule::SomeModule() {}
void SomeModule::start() { ... }
// ...
```

```
# Before (cmake file)
file(SOMEMODULE_SOURCES "SomeModule.cpp")
add_library(somemodule SHARED ${SOMEMODULE_SOURCES})
target_link_libraries(somemodule PUBLIC DaqlingCore
↪  DaqlingUtils ...)
```

The exported `daqling_*` functions seen here are required at runtime so that DAQling knows how to construct and destruct a module instance, but they are overhead; we would rather have them be automatically declared for the author. And just so we now expose DAQling-specific CMake macros for that end:

```
// After (source file)
SomeModule::SomeModule() {}
SomeModule::SomeModule() {}
void SomeModule::start() { ... }
// ...
```

```
# After (cmake file)
daqling_module(module_name)
daqling_target_sources(${module_name} SomeModule.cpp)
```

In the background, a templated source file is processed by CMake for each module and added to its list of source files, resulting in the same code in the end.

Related merge requests: !22.

2.3 Overhauled Logging Implementation

DAQling offers a very simple logging API. Without having to pass around some logger instace, an author can write `INFO("something informational")`, `WARNING("some warning")`, `DEBUG("a debug entry")` or similar from anywhere using the many available logging macros. Course, these macros all expand into writes to some global logger instance which is responsible for printing the strings to whatever destination is configured (the terminal, the system log, e.g.).[1] For example, one may have the following: a function that exists inside of a module, and one that exists outside of it:

```
void outside_module()
{
    INFO("something informational")
    std::string str = "msg";
    WARNING("component X crashed: " << str);
}

void SomeModule::start()
{
    DEBUG("starting up...");
}
```

During runtime, the following log entries are then made:

```
[16:11] [info] [outside_module()] something
↪  informational
[16:11] [warning] [outside_module()] component X
↪  crashed: msg
[16:11] [debug] [SomeModule::start()] starting up...
```

The problem with this implementation is that there is no way to telling DAQling to swallow all log entries that do not originate from, say, a module under debug; if you tell DAQling to log debug entries, a developer will also be bombarded with log entries from other DAQling components, code that they may not have any interest in. The behavior we seek is that log entries done inside a module are categorized under "module", and any logs externally under "core" (effectively, we wish to have a module logger instance, and a core logger instance, respectively):

[1]This static logger instance is wrapped in a class so that we can apply own own rules of access to it. In that sense, this instance is not a "true" global.

```
[16:11] [core] [info] [outside_module()] something
↪  informational
[16:11] [core] [warning] [outside_module()] component
↪  X crashed: msg
[16:11] [module] [debug] [SomeModule::start()]
↪  starting up...
```

Previously, this single global logger instance *symbol* existed in the built `daqling` executable. During runtime when the module is dynamically loaded, any functions that log (i.e., refer to this logger symbol) are "told" to log entries to the `daqling` symbol. By adding the same symbol inside of the shared library built from a module implementation (and hiding it from external view; keeping it off the symbol table), any log entries made in a module will instead be done to the module-internal symbol (the module logger). Log entries made elsewhere are instead made to the `daqling` symbol (the core logger). Hiding the module symbol is required: by default all symbols are visible in the symbol table, and when `daqling` dynamically loads the module it will find the module's symbol, which has the same name as its own logger symbol, and *alias* the module's symbol to its own. We must then hide the module's symbol if we wish to keep two logger instances while using the same symbol name.

In summary: the logging implementation now links logging entries made in a module to a module logger, and all other log entries to a core logger, while keeping the same API.

Related merge requests: !27.

2.4 Custom Module Commands

DAQling modules can now register custom commands during its configuration step that can be run from the DAQ interface. Currently, only commands with the signature `void(const std::string&)` are supported, but support for the signatures `void()`, `void(int)` is in progress:

```
void SomeModule::configure()
{
    // ...

    // Currently supported
    registerCommand("stringCommand", [](const
    ↪  std::string &str) {
```

```
        INFO("Got argument '" << str << "'");
    });

    // Support in-progress
    registerCommand("voidCommand", []() { /* ... */ });
    registerCommand("intCommand", [](int) { /* ... */
    ↪  });

}
```

From the DAQ interface, one may then execute `command stringCommand foobar` for the string `"Got argument 'foobar'"` to be logged.

Related merge requests: !43, !46.

2.5 Other DAQling Changes

Aside from the feature implementations above, DAQling's code base has seen varied changes in other sections:

- `Utils/ReusableThread`: improved API for task assignment; fixed a fatal race condition during destruction; and now ensures that all assigned tasks are in fact executed. See !25 and !42, respectively.
- `Utils/Binary`: reimplemented using `std::vector`; up to ~3x faster for key operations; `xxd(1)`-like string representation of binary content. See !31.
- `Core/PluginManager`: added extensive error handling. See commits `36f4b370`, `e03ff9d0` and !37.
- `Core/Command`: simplified implementation. See !47.
- `Core`: fixed a fatal race condition during module destruction where resources would be freed before the thread using them had been joined. See !34.
- Continuous integration: added configuration file for continuous integration on GitLab: all merge requests must first pass compilation, execution of any tests and a formatting check before they can be merged. See !41.

References

[1] Kevin Thomas Bauer. *FELIX: the new detector readout system for the ATLAS experiment*. Tech. rep. ATL-DAQ-PROC-2017-042. Geneva: CERN, Nov. 2017. URL: http://cds.cern.ch/record/2292375.

[2] CERN. *NA62*. URL: https://home.cern/science/experiments/na62. (accessed: 2019-08-26).

[3] Gamberini Enrico and Sonesten Viktor Vilhelm. *NA62 DAQ Upgrade: concept L1-responder*. URL: https://gitlab.cern.ch/snippets/921. (accessed: 2019-08-26).

[4] Gamberini Enrico, Sonesten Viktor Vilhelm, Sipos Roland, et al. *DAQling: Software framework for development of modular and distributed data acquisition systems*. URL: https://gitlab.cern.ch/ep-dt-di/daq/daqling. (accessed: 2019-08-26).

[5] Sonesten Viktor Vilhelm. *bin2felix*. URL: https://gitlab.cern.ch/ep-dt-di/daq/na62/tdc-simulator/blob/efd10ebf0f09467d4bab87724ae6d029577501c2/bin2felix.cpp. (accessed: 2019-08-26).

[6] Sonesten Viktor Vilhelm, Boretto Marco, and Gamberini Enrico. *process_hits*. URL: https://gitlab.cern.ch/ep-dt-di/daq/na62/tdc-simulator/blob/efd10ebf0f09467d4bab87724ae6d029577501c2/process_hits.cpp. (accessed: 2019-08-26).