World Academy of Science, Engineering and Technology
International Journal of Physical and Mathematical Sciences
Vol:11, No:12, 2017

The DAQ Debugger for iFDAQ of the COMPASS Experiment

Y. Bai, M. Bodlak, V. Frolov, S. Huber, V. Jary, I. Konorov, D. Levit, J. Novy, D. Steffen, O. Subrt, M. Virius

***Abstract*—**In general, state-of-the-art Data Acquisition Systems (DAQ) in high energy physics experiments must satisfy high requirements in terms of reliability, efficiency and data rate capability. This paper presents the development and deployment of a debugging tool named DAQ Debugger for the intelligent, FPGA-based Data Acquisition System (iFDAQ) of the COMPASS experiment at CERN. Utilizing a hardware event builder, the iFDAQ is designed to be able to readout data at the average maximum rate of 1.5 GB/s of the experiment. In complex softwares, such as the iFDAQ, having thousands of lines of code, the debugging process is absolutely essential to reveal all software issues. Unfortunately, conventional debugging of the iFDAQ is not possible during the real data taking. The DAQ Debugger is a tool for identifying a problem, isolating the source of the problem, and then either correcting the problem or determining a way to work around it. It provides the layer for an easy integration to any process and has no impact on the process performance. Based on handling of system signals, the DAQ Debugger represents an alternative to conventional debuggers provided by most integrated development environments. Whenever problem occurs, it generates reports containing all necessary information important for a deeper investigation and analysis. The DAQ Debugger was fully incorporated to all processes in the iFDAQ during the run 2016. It helped to reveal remaining software issues and improved significantly the stability of the system in comparison with the previous run. In the paper, we present the DAQ Debugger from several insights and discuss it in a detailed way.

***Keywords*—**DAQ debugger, data acquisition system, FPGA, system signals, Qt framework.

I. Introduction

THE DAQ Debugger is a tool to detect, investigate and fix software problems in the iFDAQ. It has become an essential component of the iFDAQ during the run 2016 and 2017 and helped to improve the stability of the iFDAQ significantly. In general, it is considered as an useful tool for bugs identification and backward debugging. The paper describes the route of events trough development and deployment of the DAQ Debugger for the intelligent, FPGA-based Data Acquisition System (iFDAQ) of the COMPASS experiment at CERN.

O. Subrt is with the Czech Technical University, Department of Software Engineering, Prague, Czech Republic and the European Organization for Nuclear Research – CERN, Switzerland (corresponding author, e-mail: ondrej.subrt@cern.ch).

M. Bodlak, V. Jary, J. Novy and M. Virius are with the Czech Technical University, Department of Software Engineering, Prague, Czech Republic.

D. Steffen is with the Technische Universität München, Physik-Department, Munich, Germany and the European Organization for Nuclear Research – CERN, Switzerland.

Y. Bai, S. Huber, I. Konorov and D. Levit are with the Technische Universität München, Physik-Department, Munich, Germany.

V. Frolov is with the Joint Institute for Nuclear Research, Dubna, Moscow region, Russia.

The purpose of the COMPASS (Common Muon and Proton Apparatus for Structure and Spectroscopy) experiment [1] is the study of nucleon spin structure and hadron spectroscopy. The experiment, which utilizes a polarized target and is situated at the Super Proton Synchrotron (SPS) at CERN in Geneva, Switzerland, was approved conditionally in 1997 and commissioned in 2001. In 2002, the experiment started operating and has since been making use of the various particle beams available at the CERN M2 beam line, primarily the muon and hadron beams. Typical data rate of this experiment is approximately 1.5 GB/s during 10 seconds on-spill, the off-spill time varies between 30 and 50 seconds depending on SPS super cycle.

In 2010, an extension of the experiment, COMPASS-II [2], has been approved. Based on the approval, the COMPASS-II program consists of two physics programs – the polarized Drell-Yan (DY) process in 2014, 2015 and 2018 and Deeply Virtual Compton Scattering (DVCS) in 2016 and 2017.

The paper is organized as follows. In Section II, an overview of the essential hardware technologies used in the iFDAQ is provided, followed by a description of the hardware structure of the iFDAQ and the role of the technologies in it. Moreover, the section gives an overview of the software technologies and the software structure.

Section III deals with the exact definition of debugging. It describes steps from finding to resolving of defects that prevent correct operation of computer software or a system.

Section IV is concerned with the implementation of the DAQ Debugger. It starts with the library description and discusses the integration procedure to any process in a detailed way. The implementation part and scenarios give more information about inner mechanisms of the library.

II. iFDAQ Architecture

COMPASS is in operation since 2002. Since then the amount of collected data is steadily increasing. The major growth was caused by improvement of the beam intensity and increase of trigger rates and it is supposed to continue to rise in future. Over the years, the electronics of the DAQ was upgraded several times in order to be able to handle such amount of data. Furthermore, the hardware upgrades were getting more and more complicated due to obsolete technology. Consequently, the COMPASS collaboration has decided for the considerable iFDAQ improvement during the shutdown in 2013/2014. Nowadays, the final part of hardware and software replacement finishes.

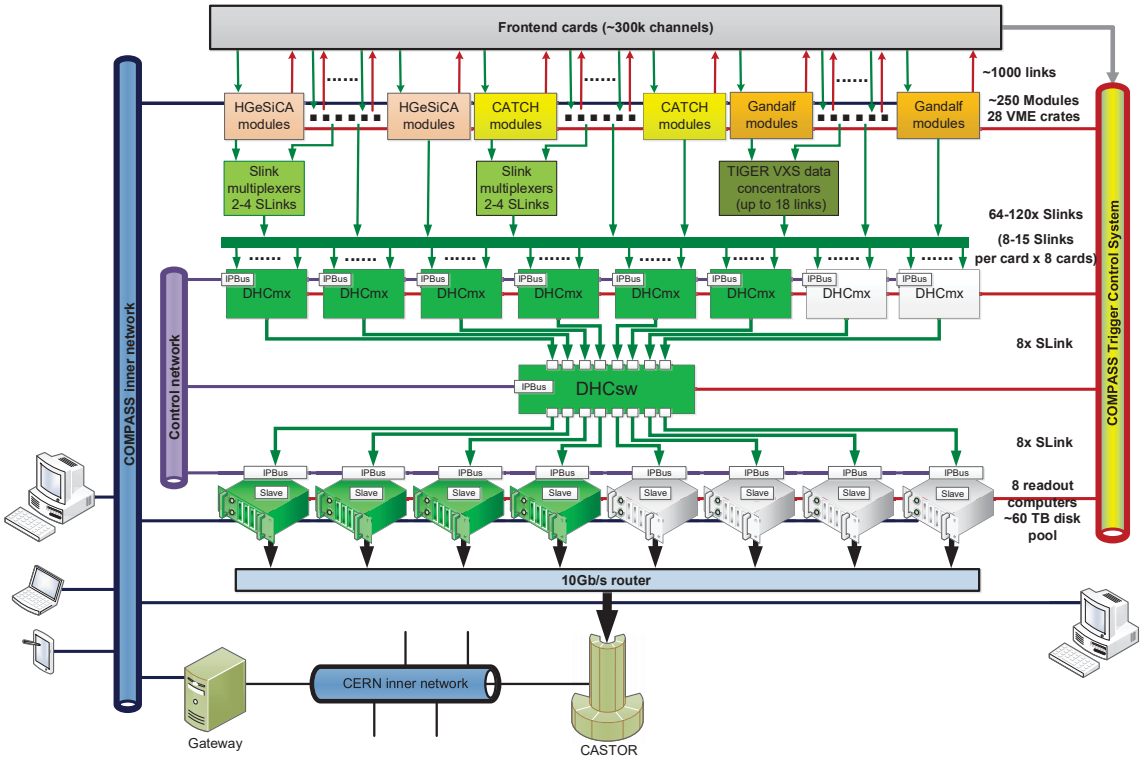

Fig. 1 The COMPASS iFDAQ topology

A. Hardware Part

The iFDAQ of the COMPASS experiment can be divided into five basic layers [9], [11], [12], the first one being frontend cards which process analog data from 300 000 detector channels and convert them to digital form. A custom timing and trigger distribution system TCS [3] (Trigger Control System) uses optical links to provide a unique event identification and time synchronization to the detectors frontends, which append these information to the digitized signal. The frontend cards are connected to data concentrator modules modules (HGeSiCA, CATCH and Gandalf) which make up the second layer. The second layer handles the first level of multiplexing (consolidating multiple data streams into a single stream). More information about HGeSiCA, CATCH and Gandalf modules can be found at [15]-[17], respectively. The data from some of the HGeSiCA and CATCH modules go through S-Link multiplexers and the data from Gandalf modules through TIGER VXS data concentrators, creating a sublayer.

Using S-Links [19], this sublayer is connected to the third layer, which comprises eight FPGA cards (DHCmx) which are called Data Handling Cards (DHC). The third layer handles another level of multiplexing. S-Links are also used to connect the third layer to the fourth layer, which is made up of a single DHC with switch firmware (DHCsw) - this layer handles event building. The fifth layer, again utilizing S-Links for connection to the previous layer, consists of eight readout computers which run the iFDAQ software. These computers are collectively referred to as the readout engine. The connection of an S-Link and the memory of a readout computer is handled by a Spillbuffer – a PCI Express card with an FPGA chip and 2GB RAM, which is also partially used for buffering. The acquired data which are to be stored are then sent directly to the CERN CASTOR facility [14]. All DHC cards are configured and controlled over separate network by processes using the IPBus [13] based on configuration in XML connection files and address files.

Online data verification and consistency check is performed by the proposed hardware builder itself. In sum, the hardware builder forms more comprehensive control system.

In Fig. 1, the current state – used in the run 2016 and 2017 – is given. It consists of only six FPGA cards (DHCmx) on the level of multiplexing and four readout engine computers.

B. Software Part

Major parts of the iFDAQ software has been implemented in C++ language. It is supported by MySQL for database access and Python with bash scripts for minor tasks. PHP, HMTL, javascript and AJAX technologies have been used for development of web-based configuration interface. The Qt framework, a cross-platform application framework, has been used for all main graphical user interfaces (GUIs) and to speed up development of core applications. The iFDAQ software [12] is deployed on the readout engine, the individual computers of which run the Scientific Linux CERN 6 (SLC6) operating system [18].

The DIALOG library [4] (distributed, inter-process, asynchronous, library, open, general) is used for

communication between processes of the iFDAQ. The DIALOG library is a multi-platform library that serves for an asynchronous one to many communication through the Ethernet.

In complex softwares, such as the iFDAQ, all features must be divided into several processes. Then each process is responsible for its intended purpose and its proper behaviour is absolutely essential for the overall system stability.

There are six types of processes fulfilling main functions in the iFDAQ [10]:

- **Master process** – The Master is a vital process for the iFDAQ - using DIALOG, it mediates communication between the Runcontrol GUI and the slave processes as well as the communication between the slave processes and the configuration database. It also plays a major role in the iFDAQ's error handling.
- **Slave-control** – The purpose of the Slave-control process is to configure and monitor the FPGA cards - it is the only process which communicates with the FPGA cards directly. All communication with the FPGA cards is carried out using IPbus.
- **Slave-readout** – The Slave-readout is a very resource-demanding process responsible for readout of data from connected devices, as well as its processing and subsequent storage. It comprises a large number of threads.
- **Runcontrol GUI** – The Runcontrol GUI, which can run in two different modes, is the means of user interaction with the iFDAQ. The first mode, Runcontrol, provides the user with complete control over the iFDAQ as well as information concerning the current run and status of the hardware. Only one instance of this mode can run at a time. The second mode, Monitoring, retains the information and status providing capabilities, but does not provide the user with any direct control over the iFDAQ. There is no practical limit to how many instances of this mode can run at a time.
- **MessageLogger** – A process that receives informative and error messages and stores them into the MySQL database. It is directly connected to the Master and to the slave processes via the DIALOG library.
- **MessageBrowser** – A GUI application that provides an intuitive access to messages from system (stored in the database) with an addition of online mode (displaying new messages in realtime). Equipped with filtering and sorting capabilities, it is able to run independently from the whole system in case of emergency.

In the original DAQ, the DATE (Data Acquisition and Test Environment) package [5] was responsible for all related data acquisitions tasks such as configuration, run control, load balancing, readout, event building, etc. Now, part of it is implemented directly in the firmware of FPGA cards. Since tools for data quality monitoring or for data analysis are based on DATE format, the system must support the same data format as defined by DATE. Transformation of read out data to DATE format is needed in order to ensure full compatibility of the iFDAQ with older COMPASS tools.

C. The Motivation for the DAQ Debugger Implementation

The iFDAQ faced several crashes of Master process and Slave-readout process per day in the runs 2014 and 2015. Processes crashed without any obvious reason or additional information.

The possibility of conventional debugging during the real data-taking is quite limited.

- It would waste the beam time during crash investigation.
- The performance of debugged processes would be lower.
- The conventional debugging process would increase load on readout engine computers.
- The iFDAQ expert would have to be present 24x7 on site.

In sum, the conventional debugging is possible only during so called machine development, i.e., time without beam. Unfortunately, the errors do not occur without the real data-taking and all processes are running smoothly. Under the above mentioned circumstances, it gets caught in a vicious circle. Conventional debugging is not usable and effective for the error detection.

The iFDAQ group made a decision to implement their own DAQ Debugger. It should help to detect remaining software issues and improved significantly the stability of the system.

III. Conventional Debugging

In computer programming and engineering, debugging [6]-[8] is a multistep process that involves attempt to reproduce the problem and isolating the source of the problem. Then the second phase of fixing the problem follows. It can be either fully corrected or determined a way to work around it. The final step of debugging is a verification that the fix works and nothing else is broken.

Once an error has been identified, it is essential to detect the error in the source code. Integrated development environment (IDE) is usually very useful in error detection. The state-of-the-art IDEs provide developers with a stand-alone debugger tool or the debugging component helping developers to find the error in the source code.

The standard debugging tool provides the programmer with the capability to examine program states (values of variables, call stack, etc.) and track down the origin of the problem. The control of program execution is assured by setting up a "breakpoint" and run the program until that breakpoint. Once the program meets any breakpoint, the program execution stops and waits. The control of program execution also offers to execute just the next line of code, step into the body of function/method or even change the value of variables.

In software development, debugging is part of the software testing process and is an essential part of the entire software development life cycle. The debugging process starts as soon as a release candidate is implemented and continues step by step to form a final version of software.

IV. DAQ Debugger

The DAQ Debugger is a library helping with the iFDAQ error detection. The DAQ Debugger was fully incorporated to all processes in the iFDAQ during the run 2016 and 2017. In general, the integration is very simple to any process. The

main goal is to produce a report concerning the process crash. The report must contain as much information as possible. Afterwards, the reports are investigated by iFDAQ experts trying to detect the source of problem. After understanding of a problem, the fix is released and tested. The DAQ Debugger is designed in order to meet the following requirements:

- The integration to running system requires interface for an easy use.
- It does not affect the process performance.
- It does not increase load on readout engine computers.
- It provides with reports in `/tmp` folder containing stack trace of all threads and memory dump.

A. Description

The DAQ Debugger is a library easily integrated to a process and standing in the background of a running process. If the process is running without any crashes, basically, the DAQ Debugger is only part of the process without any action taken and behaves during the whole process life cycle in this way.

At the operating system level, the fault is caught and a signal is passed on to the offending process, activating the process's handler for that signal. Different operating systems have different signal names to indicate that a fault has occurred. For instance, in case of a segmentation violation, a signal called SIGSEGV (abbreviated from segmentation violation) is sent to the offending process on Unix-based operating systems.

The main idea of action taken in the right instant is based on catching of system signals (SIGSEGV, SIGABRT, etc.). In case of a process crash, the following procedure is started:

- The system signal is caught and forwarded to a signal handler in the DAQ Debugger.
- The memory dump is produced and stored.
- The whole stack trace for each thread is generated with file names and code line numbers.
- The report containing the caught signal and stack trace for each thread is created in `/tmp` folder.
- The process is exiting with the caught signal.

B. Integration

The DAQ Debugger is designed bearing in mind that it has to be integrated in a running system, so it has to be made as easy to use as possible. To incorporate it to any process, the static initialization method is called in a single line, as you can see in the following example showing the integration of the DAQ Debugger into a Qt-CoreApplication.

```
#include <QCoreApplication>
#include "daqdebugger.h"

int main(int argc, char **argv)
{
    QCoreApplication* app = new QApplication(argc, argv);
    DAQDebugger::init(argv[0]);
    return app->exec();
}
```

The first argument of `DAQDebugger::init(argv[0])` is the name of a process. Then the name of a process is included in the report file name.

Since the DAQ Debugger is a library, it must be located on the system path. Generally, `LD_LIBRARY_PATH` is used to specify directories of libraries. It is also necessary to add the following lines to the Qt-project file (*.pro) in order to create the makefile correctly.

```
INCLUDEPATH += PATH_TO_DAQ_DEBUGGER/
DEPENDPATH += PATH_TO_DAQ_DEBUGGER/
LIBS += -L PATH_TO_DAQ_DEBUGGER -lDAQDebugger

QMAKE_CXXFLAGS += -rdynamic -g
QMAKE_LFLAGS += -rdynamic -g

QMAKE_CXXFLAGS +=
-include PATH_TO_DAQ_DEBUGGER/qthreaddaqdebugger.h
QMAKE_CXXFLAGS +=
-include PATH_TO_DAQ_DEBUGGER/qthreaddaqdebugger_macro.h
```

GCC flags `-rdynamic` and `-g` enable use of extra debugging information. The `-rdynamic` option instructs the linker to add symbols to the symbol tables that are not normally needed at run time. The `-g` option produce debugging information in the operating system's native format.

The `-include` option processes file as if `#include "file"` appeared as the first line of the primary source file. If multiple `-include` options are given, the files are included in the order they appear on the command line. Demand on `-include` statements is discussed in the implementation subsection in a deeper way.

The process should be compiled without any optimizations, e.g, `-O`, `-O1`, `-O2`, `-O3` or `-Os`, if possible. The default value usually is `-O0` that means do not optimize. It is important for `addr2line` command that is used for detection of an exact file name and a line number crash.

Using optimizations, the compiled source code could be inline and detection of an exact file name and a line number crash is then much harder. Hence using `addr2line` command on processes compiled with optimizations could lead to shifted or mistaken line numbers.

On the other hand, optimizations are very useful for compilation of libraries due to libraries' shared and linking purpose.

To sum up, turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

C. Implementation

The system signals to catch are specified in the `DAQDebugger::init(argv[0])` static method. By default, signals SIGABRT, SIGSEGV, SIGILL and SIGFPE are registered. The signals to catch can be added or removed there.

Whenever the registered signal is caught, it is caught in the thread causing the crash and the stack trace of thread can be easily produced. It could be enough in single-threaded processes. Unfortunately, the solution must be more general and considered even multi-threaded processes. The solution must provide the following crash procedure:

- The system signal is caught in the crashed thread.
- All remaining threads are immediately suspended.
- Store memory dump.
- Get stack trace of the crashed thread.
- Get stack traces of suspended threads.

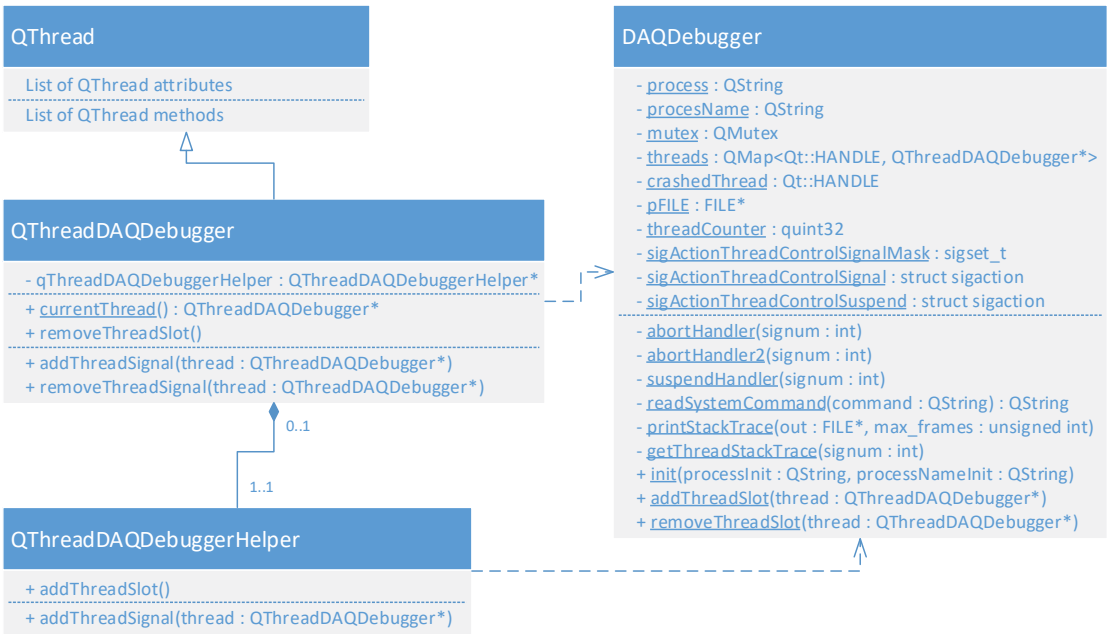

Fig. 2 Class diagram of the DAQ Debugger

- The crashed thread (whole process) is exiting with the caught signal.

To register a system signal, the following statement is executed.

```
// to register a system signal
signal(signal, signalHandler);
```

POSIX [20] defines a standard threading library API in order to control and send suspend/resume signals to threads. All important statements are given in the following source code with the explanations in comments.

```
// to send a signal to thread ID
pthread_kill((pthread_t)threadID, signal)

// to catch the sent signal in a thread
struct sigaction sigActionThreadControlSignal;
sigfillset(&sigActionThreadControlSignal.sa_mask);
sigdelset(&sigActionThreadControlSignal.sa_mask, signal);

sigActionThreadControlSignal.sa_flags = 0;
sigActionThreadControlSignal.sa_handler = signalHandler;
sigaction(signal, &sigActionThreadControlSignal, NULL);

// to suspend a thread
sigset_t sigActionThreadControlSignalMask;
sigfillset(&sigActionThreadControlSignalMask);
sigdelset(&sigActionThreadControlSignalMask, signal2);

sigsuspend(&sigActionThreadControlSignalMask);
```

The `signalHandler` is registered in `sigaction` and it is triggered if the `signal` is sent to the thread. Moreover, the thread is suspended by `sigsuspend` with given signal mask and it resumes if the `signal2` is sent to the thread.

At this point, the control of threads is prepared, the DAQ Debugger can obtain the stack trace with file names and line numbers for each thread. Using `backtrace` and `backtrace_symbols`, the stack trace is generated. The `backtrace` command returns the series of currently active function calls for the process. Moreover using `backtrace_symbols`, the symbolic description of function calls is translated from information obtained by `backtrace` to function names and hexadecimal addresses. Unfortunately, it returns each line of stack trace in a hexadecimal address format and thus it is not easily readable for a human being. However, to overcome this drawback, the DAQ Debugger is using `addr2line`. It is capable to convert hexadecimal addresses into file names and line numbers. In the following code, you can see a short example.

```
// storage array for stack trace address data
unsigned int max_frames = 63;
void* addrlist[max_frames + 1];

// retrieve current stack addresses
unsigned int addrlen = backtrace(addrlist, sizeof
(addrlist)/sizeof(void*));

// resolve addresses into strings containing
"filename(function + address)"
char** symbollist = backtrace_symbols(addrlist, addrlen);

for (unsigned int i = 1; i < addrlen; i++)
    std::cout << readSystemCommand
    ("addr2line -e " + processName + " " + getHexAddress
    (symbollist[i])) << std::endl;
```

The stack trace is one thing, on the other hand, it is still not sufficient for the error detection. To satisfy the comprehensive understanding of crash, the memory dump is absolutely essential. The DAQ Debugger is using `gcore` command for memory dump storage.

Another challenge in the design of the DAQ Debugger is the registration of all threads without violating the concept of easy integration. In order to be able to send signals and to control threads in case of a crash, it is necessary to obtain all

thread IDs at the beginning of a process.

Unfortunately, it is not an easy task to obtain IDs of all threads in a process. Moreover, it is even more complex if the integration of DAQ Debugger should be as simple as possible. The only way how to get thread ID is executing the part of code in this thread asking for thread ID. So, it must be ensured the execution of `DAQDebugger::addThreadSlot(thread)` static method in each thread. Each thread must register itself in the DAQ Debugger immediately when it starts its execution. It uses `started()` signal in `QThread` object being emitted when the thread starts executing. The functionality of `QThread` object must be extended in order to cover the registration in the DAQ Debugger and its integration would be still simple to any process. This extension is hidden and added by `-include` statements to Qt-project file (*.pro). File `qthreaddaqdebugger.h` contains the definition of thread satisfying the required functionality and inheriting from `QThread` object. Moreover, file `qthreaddaqdebugger_macro.h` replace all `QThread` objects for `QThreadDAQDebugger` objects by preprocessor definition in the whole process as follows.

```
#define QThread QThreadDAQDebugger
```

In Fig. 2, you can see the DAQ Debugger class diagram being obtained by the described integration process.

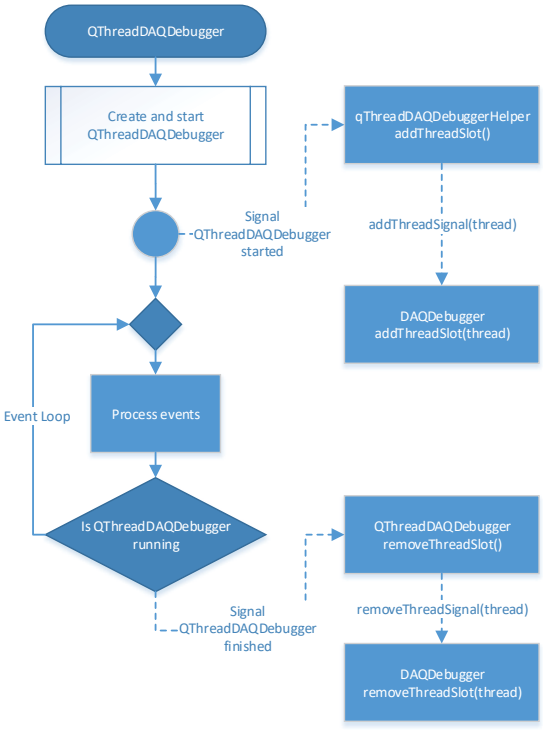

Fig. 3 Flow diagram of the thread life cycle in the DAQ Debugger

The way how to introduce the DAQ Debugger to each thread has been described. It remains to discuss all most common scenarios, including the thread registration procedure, in the DAQ Debugger in a deeper way. It is given in a next subsection.

D. Scenarios

Basic scenarios are emphasized in this subsection dealing with how one or more components interact inside the DAQ Debugger or with the DAQ Debugger itself.

The description of thread life cycle gives a comprehensive insight from a global point of view. The diagram is shown in Fig. 3. The `QThreadDAQDebugger` object inheriting from `QThread` object is created and the thread is started. The signal `started()` is emitted and it is connected to the slot `addThreadSlot()` of `qThreadDAQDebuggerHelper` object. This object has been already moved to the thread of `QThreadDAQDebugger` object in the `QThreadDAQDebugger` object constructor since it must live in this thread. This is the way how to force the execution of `DAQDebugger::addThreadSlot(thread)` static method in this thread and thus the DAQ Debugger gets thread ID. Finally, the slot `addThreadSlot()` of `qThreadDAQDebuggerHelper` object is emitting the signal `addThreadSignal(thread)` being connected to the `DAQDebugger::addThreadSlot(thread)` static method. This concept ensures the execution of `DAQDebugger::addThreadSlot(thread)` in our thread.

Moreover, Fig. 3 covers the concept when a thread is finishing its execution. The thread must unregister in the DAQ Debugger. When the `QThreadDAQDebugger` object finishes its execution the signal `finished()` is emitted and it is connected to the slot `removeThreadSlot()` of `QThreadDAQDebugger` object. There the signal `removeThreadSignal(thread)` is emitted and it goes directly to the `removeThreadSlot(thread)` of DAQ Debugger. In comparison with the registration procedure, the unregistration procedure is much simpler since the DAQ Debugger already knows the finished thread. Since, the `QThreadDAQDebugger` object lives in the main thread no one has to worry about handling of emitted `QThreadDAQDebugger` signals after the thread has finished its execution. These emitted signals are handled by the main thread.

To finish the discussion concerning the thread registration procedure properly, the description of registration of $n \in \mathbb{N}$ threads when a process starts is given. In Fig. 4, the diagram begins with the main thread. First of all, the main thread starts its execution. It registers system signals and registers the main thread in the DAQ Debugger. Whole mentioned functionality is encapsulated in the `DAQDebugger::init(argv[0])` static method. Then the main thread continues its execution and processes events. All remaining of $n \in \mathbb{N}$ threads are registered in the DAQ Debugger afterwards. The detail of registration procedure for each thread was already mentioned and it triggered with the emitting of signal `started()`. Of course, whenever some of $n \in \mathbb{N}$ threads finish their execution, they are unregistred from the DAQ Debugger. For simplicity

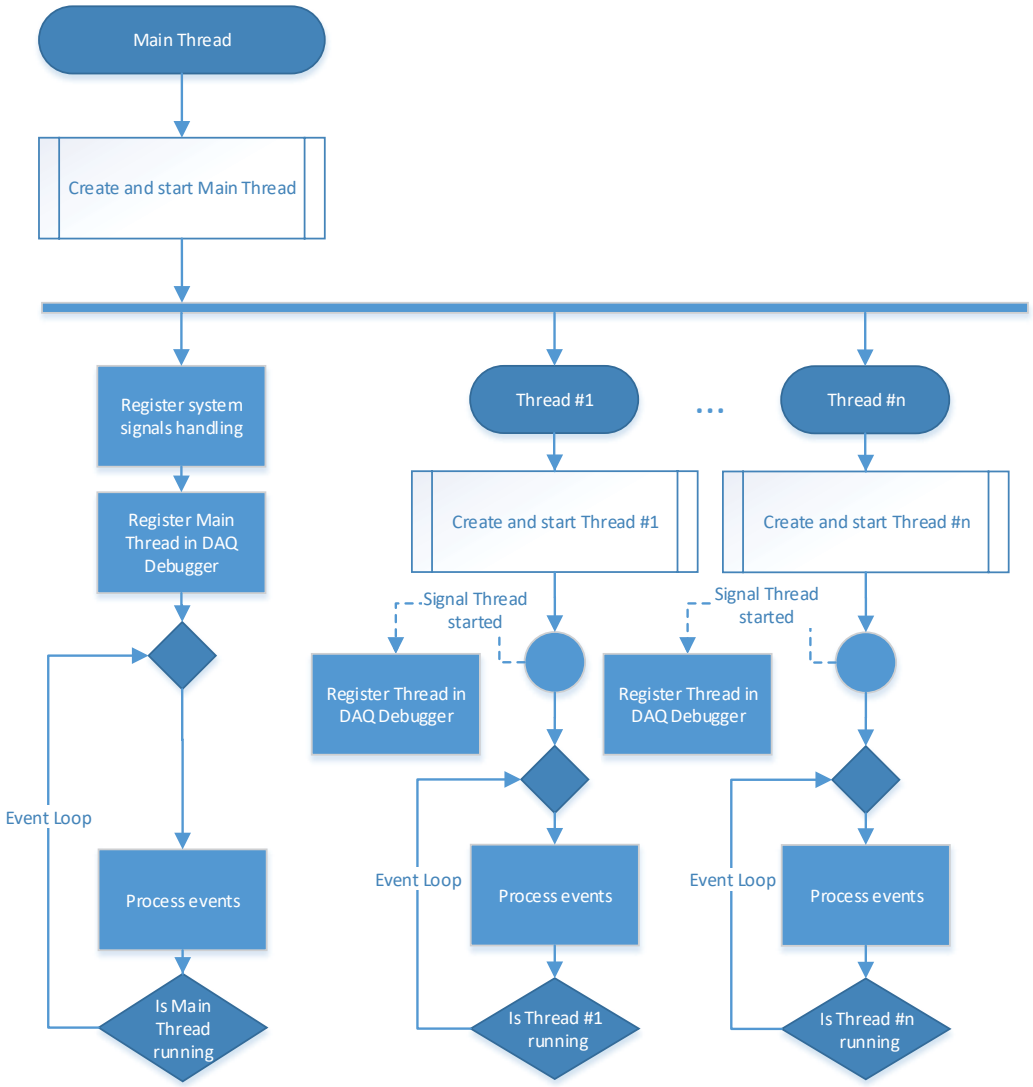

Fig. 4 Flow diagram of the thread registration procedure in the DAQ Debugger

reasons, the unregistration procedure is not depicted in the diagram.

Probably the most important scenario is the crash of a process. This situation is described in Fig. 5. From a process start, the DAQ Debugger is a part of a process and standing in the background of a running process. If the process is running smoothly without any single crash, the DAQ Debugger does not take any action. For this reason, the DAQ Debugger does not affect the process performance and does not increase load on readout engine computers at all.

The system signals are registered, the process continues its execution. Once the crash of process occurs, the DAQ Debugger handles it. The system signal is emitted and it is caught by the signal handler of crashed thread in the DAQ Debugger. At this point, it is important to realize the crashed thread where crash has occurred is responsible for the control of all remaining threads, memory dump storage and creation of crash report.

Firstly, the crashed thread sends the suspend signal to all remaining threads. It is necessary to suspend them otherwise they would continue their execution and thus the exact point of crash would be lost. Then the memory dump is produced and stored. The memory dump can be easily loaded to Qt Creator (Debug → Start Debugging → Load Core File) and memory can be investigated as much as by conventional debugging.

Secondly, the report file is created and open for writing. The crashed thread writes its stack trace to the file.

Afterwards, the control of all suspended threads is started. The crashed thread sends the resume signal to first suspended thread and the crashed thread itself is suspended. The resumed

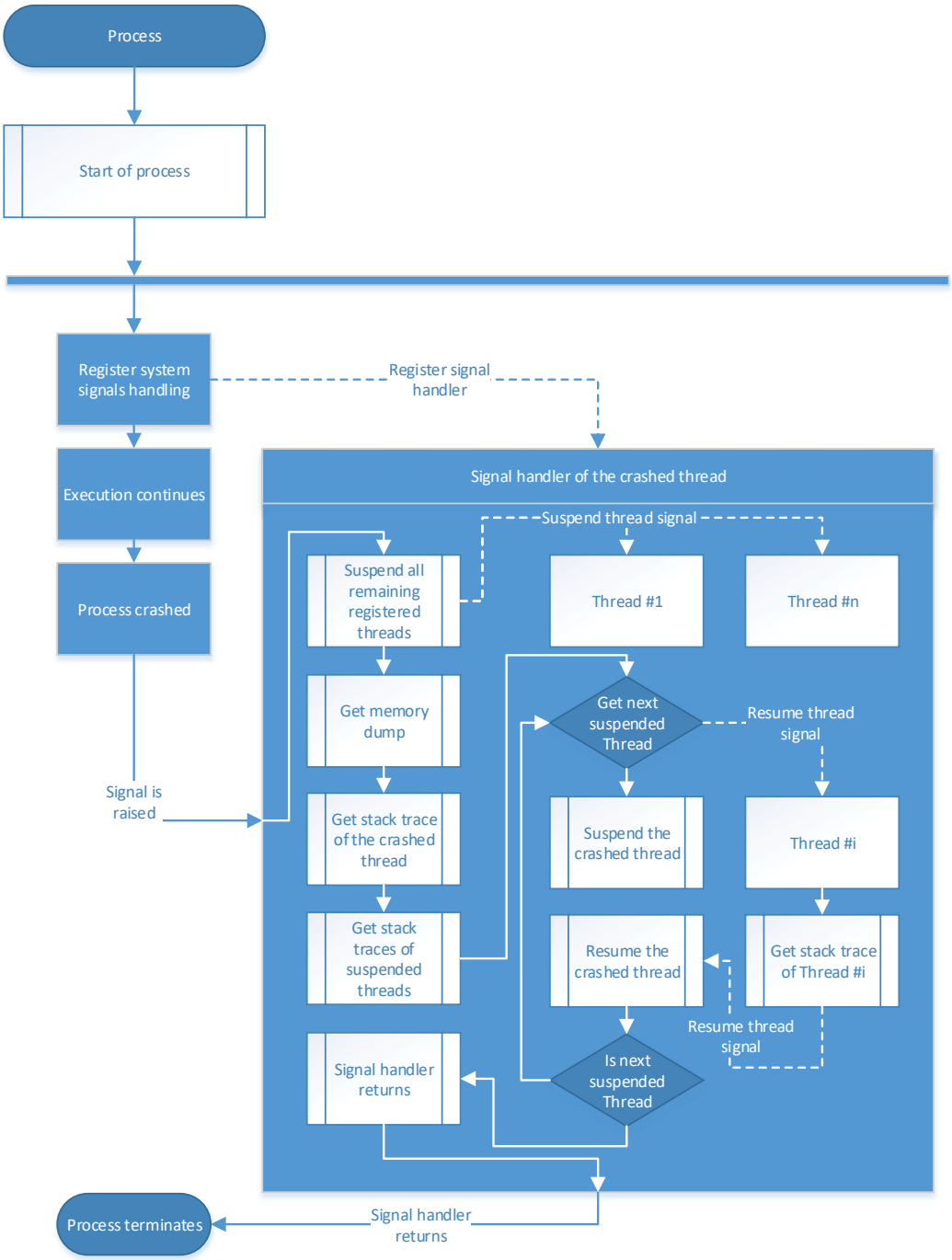

Fig. 5 Flow diagram of the thread crash caught and handled by the DAQ Debugger

thread writes its stack trace to the file, then sends the resume signal to the crashed thread and is suspended again. The resumed crashed thread sends the resume signal to second thread and it is again suspended. The second resumed thread writes its stack trace to the file, then sends the resume signal to the crashed thread and is suspended again. It continues in this way to the last suspended thread. The resumed crashed thread (resumed by the resume signal sent from $(n-1)$-th thread) sends the resume signal to n-th thread and it is again suspended. The n-th resumed thread writes its stack trace to the file, then sends the resume signal to the crashed thread and is suspended again. This suspend/resume procedure ensures

the serial writing to file and proper thread control. Finally, the report file is closed and process is exiting with the caught signal in the crashed thread. The whole control of threads, memory dump storage, opening and closing of report file is controlled by the crashed thread.

V. Conclusion

The DAQ Debugger has been incorporated to all processes of the iFDAQ in August 2016 and since then it helps with the error detection. It does not affect the process performance and does not increase load on readout engine computers.

Before the DAQ Debugger integration, the iFDAQ was facing four crashes of Mater process per day at average and several crashes of Slave-readout as well without any explanation. For this reason, to detect and resolve system crashes, the DAQ Debugger has been implemented.

Firstly, it focused on the understanding of Master process crashes. The DAQ Debugger helped significantly to detect all remaining software issues in Master process so it became stable since the end of September 2016. Since then no crash of Master process has been observed.

It improved the stability of Slave-readout as well. At the end of run 2016, the iFDAQ reached the crash rate of Slave-readout at level of one crash per four days. In July 2017, all remaining software issues in Slave-readout have been fixed. Since then the iFDAQ is stable and without any single crash.

The DAQ Debugger fulfilled initial demands and purpose and the process crash investigations based on provided crash reports continue.

References

[1] P. Abbon, et al.(the COMPASS collaboration): *The COMPASS experiment at CERN*. In: Nucl. Instrum. Methods Phys. Res., A 577, 3 (2007) pp. 455518.

[2] V. Y. Alexakhin, et al. (the COMPASS Collaboration): *COMPASS-II Proposal*. CERN-SPSC-2010-014, SPSC-P-340. May 2010.

[3] B. Grube: *A Trigger Control System for COMPASS and a Measurement of the Transverse Polarization of Lambda and Xi Hyperons from Quasi-Real Photo-Production.*. Munich. Technical University Munich. 2006. Doctoral thesis.

[4] Y. Bai, et al.: *The Communication Library DIALOG for iFDAQ of the COMPASS experiment*. 19th International Conference on High Energy Physics – ICHEP 2017, Paris, France, September 2017. International Journal of Mathematical, Computational, Physical, Electrical and Computer Engineering, vol. 11, issue 9, pp. 353-362, World Academy of Science, Engineering and Technology.

[5] T. Anticic, et al. (ALICE DAQ Project): *ALICE DAQ and ECS User's Guide* CERN, EDMS 616039, January 2006.

[6] Debugging definition. (online). Available at: http://searchsoftwarequality.techtarget.com/definition/debugging. (Accessed: 2017-09-01).

[7] T. Grötker, et al.: *The Developer's Guide to Debugging*. Second Edition, Createspace, 2012. ISBN 1-4701-8552-0.

[8] G. J. Myers: *The Art of Software Testing*. John Wiley & Sons inc, 2004. ISBN 0-471-04328-1.

[9] M. Bodlak, et al.: *Developing Control and Monitoring Software for the Data Acquisition System of the COMPASS Experiment at CERN*. Acta polytechnica: Scientific Journal of the Czech Technical University in Prague. Prague, CTU, 2013, issue 4. Available at: http://ctn.cvut.cz/ap/.

[10] M. Bodlak, et al.: *Development of new data acquisition system for COMPASS experiment*. Nuclear and Particle Physics Proceedings, 37th International Conference on High Energy Physics (ICHEP). AprilJune 2016, vol. 273275, pp. 976981. Available at: http://dx.doi.org/10.1016/j.nuclphysbps.2015.09.153.

[11] M. Bodlak, et al.: *FPGA based data acquisition system for COMPASS experiment*. Journal of Physics: Conference Series. 2014-06-11, vol. 513, issue 1, s. 012029-. DOI: 10.1088/1742-6596/513/1/012029. Available at: http://stacks.iop.org/1742-6596/513/i=1/a=012029?key=crossref.78788d23de2b4a6a34d127c361123b8c.

[12] M. Bodlak, et al.: *New data acquisition system for the COMPASS experiment*. Journal of Instrumentation. 2013-02-01, vol. 8, issue 02, C02009-C02009. DOI: 10.1088/1748-0221/8/02/C02009. Available at: http://stacks.iop.org/1748-0221/8/i=02/a=C02009?key=crossref.a76044facdf29d0fb21f9eefe3305aa5.

[13] C. Ghabrous Larrea, et al.: *IPbus: a flexible Ethernet-based control system for xTCA hardware*, 2015 JINST 10 C02019. doi:10.1088/1748-0221/10/02/C02019.

[14] CASTOR – CERN Advanced Storage manager. Available at: http://castor.web.cern.ch. [Accessed: 2017-05-01]

[15] Electronic developments for COMPASS at Freiburg. Available at: http://hpfr02.physik.uni-freiburg.de/projects/compass/electronics/catch.html. (Accessed: 2017-05-01).

[16] The GANDALF Module. (online). Available at: http://hpfr03.physik.uni-freiburg.de/gandalf/pages/information/about-gandalf.php?lang=EN. (Accessed: 2017-05-01).

[17] iMUX/HGESICA module. (online). Available at: https://twiki.cern.ch/twiki/pub/Compass/Detectors/FrontEndElectronics/imux_manual.pdf. (Accessed: 2017-05-01).

[18] Linux at CERN. (online). Available at: http://linux.web.cern.ch/linux/scientific6/. (Accessed: 2017-05-01).

[19] S-Link – High Speed Interconnect. (online). Available at: http://hsi.web.cern.ch/HSI/s-link/. (Accessed: 2017-05-01).

[20] POSIX – Standards. IEEE. (online). Available at: http://standards.ieee.org/develop/wg/POSIX.html. (Accessed: 2017-09-20).