Connectivity Implementation for ITk-DAQ-Software

Marvin Emin Geyik

University of Wuppertal
School of Mathematics & Natural Sciences

A thesis submitted for the degree of
Bachelor of Science (B.Sc.)

Main Supervisor	Prof. Dr. Wolfgang Wagner
Second Supervisor	Prof. Dr. Christian Zeitnitz

3. September 2018

Contents

List of Figures

Glossary

Acronym	Meaning
CERN	European Organization for Nuclear Research
LHC	**L**arge **H**adron **C**ollider
ATLAS	**A** **T**oroidal **L**HC **A**pparatu**S**
ITk	**I**nner **Tr**ac**k**er
DAQ	**D**ata **A**c**Q**uisition
YARR	**Y**et **A**nother **R**apid **R**eadout
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
BOC	**B**ack-**O**f-**C**rate card
ROD	**R**ead-**O**ut-**D**river
FE-chip	**F**ront-**E**nd chip
IBL	**I**nsertable **B**-**L**ayer
HW-controller	**H**ard**W**are-controller
Rx-channel	**R**eceiving channel
Tx-channel	**T**ransmitting channel

1 Introduction

The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) in Switzerland is the largest particle accelerator in the world with a circumference of 27 km. In two parallel beamlines protons are accellerated in opposite directions. These beamlines cross at four different positions, where the protons collide at record collision energies of 13 TeV [1]. At each collision point there is a cylindrically symmetric particle detector, which detects the energy, the momentum and the charge of secondary particles of the proton collisions. The largest of these detectors is the A Toroidal LHC ApparatuS (ATLAS) detector experiment. With a diameter of 25 m and a length of 46 m the ATLAS detector allows for extremely precise determination of the characteristics of secondary particles. Thus the processes that lead to the production of these particles can be analyzed in a uniquely accurate manner. Hence, the ATLAS detector and the experimental data acquiered with it are used to test theories of elementary particle physics. In 2013 the ATLAS collaboration famously published findings that indicated the discovery of the Higgs-Boson and determined its mass to 126.0(8) GeV/c^2 [2]. This discovery was an extremely important confirmation of the standard model of particle physics. Additionally, members of the ATLAS collaboration could give precise measurements of the mass of the top-quark and continue to work on searches for physics beyond the standard model.
The ATLAS detector is cylindrically shaped and constructed in layers around the collision point. The innermost part consists of a pixel detector, in the following referred to as Pixel detector, a silicon microstrip tracker and a transition radiation tracker. The entire inner detector is immersed in a magnetic field. Data acquired in this part of the detector gives precise information on a particle's trajectory and thus its momentum and charge. The middle part consists of electromagnetic and hadronic calorimeters, which determine a particle's energy. The outmost part of the detector consists of a number of muon spectrometers [3].
To further increase the amount of data generated at the LHC, the entire collider along with all detectors will undergo an upgrade, which should be completed by 2026. The aim of this upgrade is to increase the luminosity and thus the event rate of the LHC by a factor of 10. The Pixel and the silicon microstrip detector, as well as the transition radiation tracker, of the ATLAS inner detector are scheduled to be replaced in 2024 by a new tracker. The new Inner Tracker (ITk), consisting of the new Pixel and silicon microstrip detectors, needs to be able to withstand a higher dose of radiation and acquire data at a higher frequency. Thus, all components of the current inner detector need to be replaced and improved. This includes software components, in particular data acquisition software (DAQ-software) [4]. This thesis focuses on developments for the DAQ-software of the ITk Pixel detector.
The ITk-DAQ-software is based on the software of the YARR (Yet Another Rapid Readout) system [5]. This readout system is built on a basis of PCIe-cards, which connect to the on-detector electronics. It is described in more detail in Chapter 2. In this thesis, a number of new features are added to the YARR-software and their impact on the overall performance is determined. The changes are outlined in more detail in Chapter 3. The main change is the added support of a dynamic number of FPGA boards in a scan process. In addition to that, a software limitation on the number of connected detector modules per FPGA board is removed. Finally, a prototype for a custom logging class for YARR is implemented into the software.

In order to identify the implications of the changes mentioned above, the modified software is tested for performance and functionality in Chapter 4. In particular, the benefits of performing scan processes parallel on multiple FPGA boards are investigated quantitatively. Additionally, the general performance impact of output in the YARR-software is determined.

2 Contextual information

The following chapter outlines fundamental information about the Pixel detector and the YARR-software. At first, the hardware of the current Pixel detector is explained. Particularly the functionality of the specific sensors and their connection to the readout system is outlined. In Section 2.2 the ITk-DAQ-software is introduced. The functionalities important for this thesis are outlined and their implementation is explained. A quick explanation of general multithreaded programming concludes this section.

2.1 The Pixel detector at ATLAS

The ATLAS Pixel detector is the part of the ATLAS detector closest to the interaction point. The current Pixel detector can be divided into two regions. The barrel region is arranged cylindrically around the beampipe. The endcap region is located at both ends of the barrel region and has pixels, which are oriented perpendicularly to the beamline. After its completion in 2008, the original Pixel detector had over 80 million pixels. The pixel sensors are located on n-type silicon wafers. The wafers have a high positively and a high negative dopant concentration on each side and can thus be viewed as bipolar diodes. The positively doped sides of the wafers are segmented into a matrix of $400 \times 50\,\mu m^2$ pixels [6]. If an ionizing particle passes through the depletion zone of one pixel, a number of electrons are released from their atomic bond and a measurable current is produced [7].
As Figure 2.1 shows, the barrel region originally consisted of three cylindrical, concentrical layers. During an LHC shutdown from 2014 to 2015 a fourth layer, the Insertable B-Layer (IBL), was added to the Pixel detector in between the innermost layer and a new beampipe with a smaller radius. The IBL added another twelve million pixels to the detector [8].

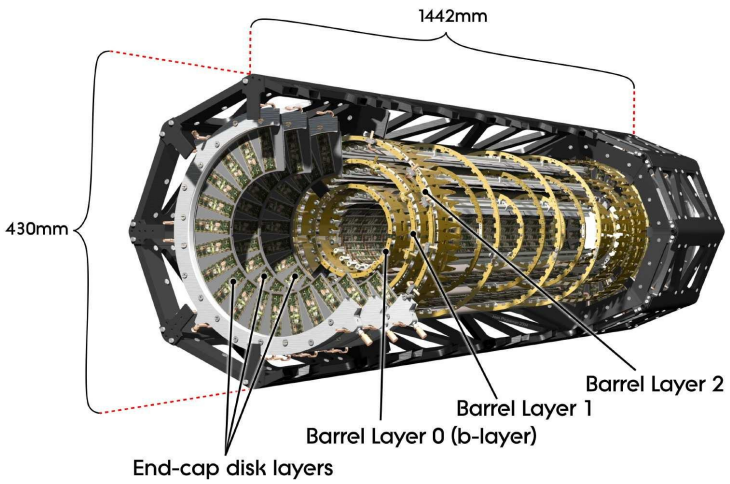


Figure 2.1: Schematic of the original ATLAS Pixel detector [7].

All four layers of the pixel detector are constructed in a similar manner. They consist of elongate mechanical/cooling supports called staves. The staves carry the pixel sensors and part of the readout electronics. There are some differences between staves of the original Pixel detector and IBL staves. Since an IBL stave is used for testing in this thesis, only IBL staves are explained in more detail in the following. More information on the staves of the original Pixel detector can be found in reference [7].
Each IBL stave carries 16 detector modules. These modules make up the smallest detector component that can be powered and operated independently. Every module consists of one silicon-sensor, which is connected to two Front-End-chips (FE-chips). These FE-chips constitute the interface between the sensor-pixels and the detector electronics. The entire pixel cell can be divided into sensor-pixel and electronic-pixel. The electronic part of the pixel is located in the FE-chip. When an ionizing particle generates a charge within a sensor-pixel cell, the amount of charge is compared to a certain threshold within the electronic-pixel cell. If the amount of generated charge exceeds the threshold, the FE-chip generates a hit, which is then sent to the readout system. Such a hit can also be generated artificially, for example by injecting a charge into the FE-chip's electronic-pixel cell. This generated charge is then compared to the threshold. Alternatively, a hit can be generated directly via a digital interface. The latter process is useful for testing the general connectivity to the FE-chip and the functionality of the DAQ system. IBL staves use FE-chips of type FE-I4B.
In general, every FE-chip has an input and an output channel. These are connected to the output and the input channels of the readout system respectively. The channel that transmits data from the readout system to the FE-chip will be called Tx-channel in the following. The channel through which the readout system receives data will be referred to as Rx-channel.
The Pixel-DAQ system of the original Pixel detector works on a basis of optical links connecting to off-detector electronics. The updated IBL-DAQ system works in a very similar manner. Every half-stave is connected to an opto-board, which sends and receives data via optical fibers from off-detector electronics. These off-detector electronics consist of two FPGA-boards, the Back-Of-Crate card (BOC) and the Read-Out-Driver (ROD). The ROD handles trigger signals sent to the FE-chips, while the BOC acts as a communicator between the ROD and the FE-chips. The FPGA on the ROD is an important part of the readout system, as it generates histograms of the received hit data. The ROD is then connected to a PC which controls and monitors the entire readout process.

2.2 The ITk-DAQ Software (YARR)

The ITk-DAQ-Software is based on the readout software of the YARR system. The system was originally developed as part of a PhD thesis by Dr. Timon Heim in 2015 (see [5]). As aforementioned, the readout system of the current Pixel detector processes some of the received data within the ROD's FPGA. This results in a complex interconnection between the readout software and firmware and makes development and improvement of the system unnecessarily difficult. The motivation behind the YARR system was to separate software from firmware for a more modular system. Furthermore, the data processing is entirely done in a PC. Thus, the FPGA-board serving as an interface between PC and on-detector electronics is relieved of all processing tasks. The only responsibility of the FPGA in the YARR system is to aggregate the received data, i.e. check the data for errors and frame it in a specific format. The YARR system is especially designed for PCIe-cards with a mounted FPGA. Their direct connection to the CPU of a PC makes them the prime choice as a YARR FPGA-board. The concept of the YARR-system in comparison to the current Pixel-DAQ-system can be seen in Figure 2.2.

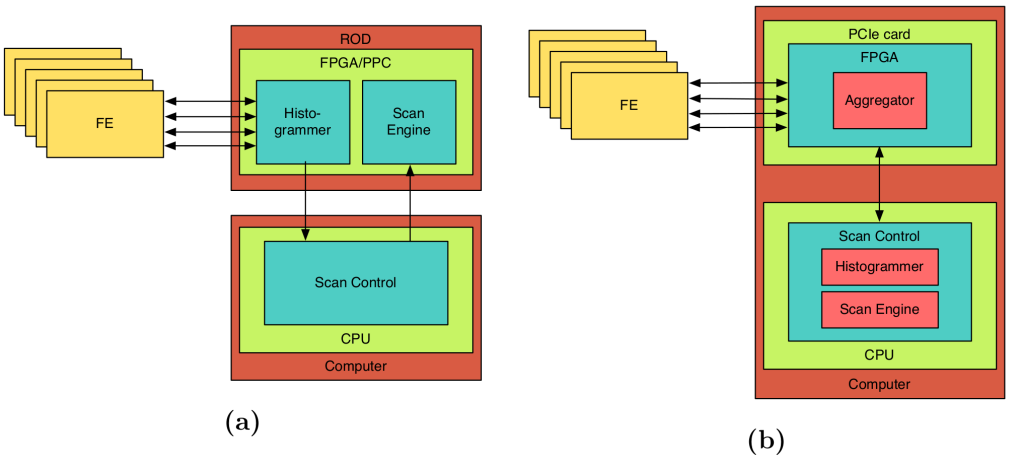


Figure 2.2: The concept of the current Pixel-DAQ-system (a) compared to the concept of the YARR-system (b) [5].

Since most of the data processing is outsourced to software components, the YARR-software is the most important and most complex part of the YARR-system. It is written in C++ and constructed in a very modular fashion, so that adding new components to the software is as easy as possible. The type of FPGA-board and the type of FE-chip connected can both be specified separately at runtime, as long as they are predefined in the software [5].

2.2.1 Scans and Loops

The specific actions that the YARR-software can perform on connected FE-chips are called scans. The simplest type of scan is the digitalscan. During the digitalscan digital hits are generated within each pixel cell of a FE-chip. These hits should then be visible in the histograms generated by the software. Hence, the digitalscan can easily serve as a test for the hardware connectivity and the general functionality of software and hardware.
There are several other scantypes defined in the YARR-software. The specific actions on the FE-chips differ among those scans, however the way in which these actions are performed remains the same.
For several reasons it is not possible to address and read out all pixels simultaneously. The chip is only able to handle a certain amount of activity. In addition, there are bandwidth limitations on the cables connected to the FE-chip and heat limitations on the FE-chips themselves. Thus, scan actions are performed in nested loops. FE-I4B chips are constructed in a way that divides the pixel matrix up into double columns. Therefore, the scan primarily masks these double columns in a certain pattern (e.g. only every fourth double column is active) and loops over all double columns. Within each active double column the pixels are masked and looped over. Then for all active pixels the scan action is performed (e.g. a charge is injected into the pixel or a digital hit is generated) a number of times. For every action the data of all active pixels is gathered. After all loops have closed the scan is finished and the histograms have been filled. Within the software, the aforementioned loops are abstract objects, which can be nested into each other. The entire set of loops is supervised by a superordinate object called the scan engine. The concept of loops and their interconnection in the YARR-software can be seen in Figure 2.3.

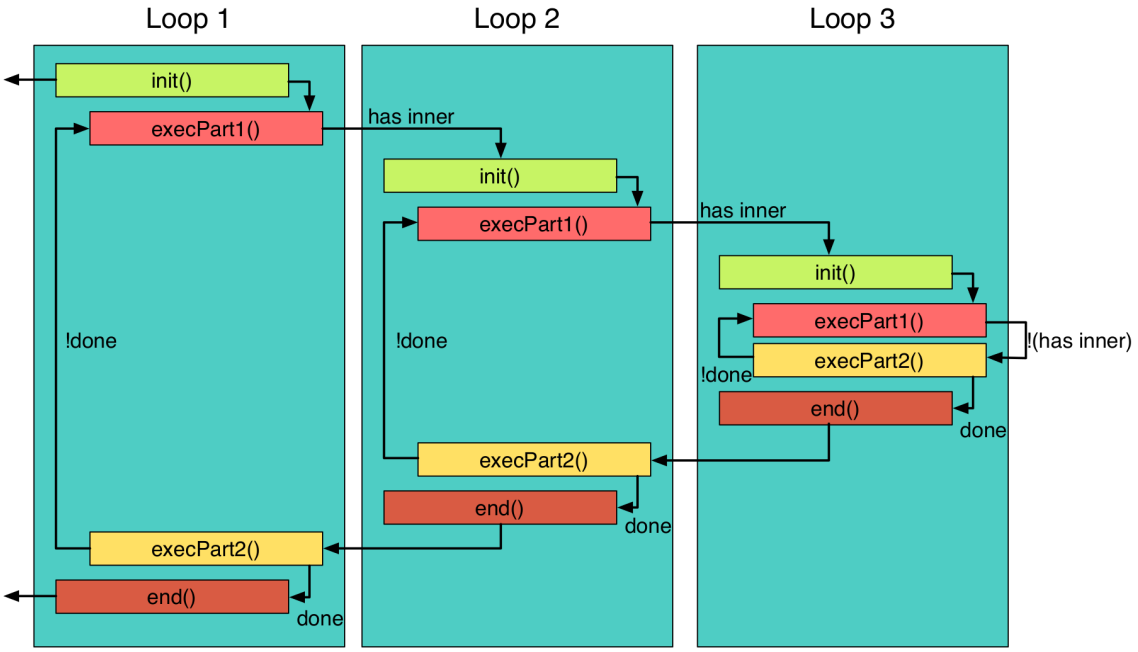


Figure 2.3: The concept of nested loops within the YARR-software [5].

The YARR-software requires a configuration file for each FE-chip to be scanned. This file includes settings for all registers of the FE-chip, like the threshold settings for every pixel and other parameters, and settings for internal analog-digital-converters. During a scan some of those parameters can be altered. After each scan a configuration file with the altered parameters is saved in the same directory as the original file.
This thesis focuses on connectivity implementations, which are best tested with the digitalscan. Thus, all functionality and performance tests are exclusively done using digitalscans. Because of the similar build up of all scans, the functionality of digitalscans should indicate a functionality of all scans, at least from the software perspective.

2.2.2 Hardware controllers

As mentioned before, the YARR-software does not specify many requirements for the FPGA-board connected to the FE-chips. Thus, the FPGA-board is implemented into the YARR-software in the form of an abstract hardware-controller (HW-controller). This HW-controller is specified via an external configuration file at runtime. In general, it is possible to specify any type of FPGA-board as HW-controller. The YARR-software, however, also provides an emulator as a HW-controller, which generates data that resembles hit data from a FE-chip. The emulator is especially useful for testing the functionality of the program without an elaborate test setup. In the current state of the software, the emulator only supports one connected FE-chip. This does not pose a problem for this thesis, however, since no tests that require more than one connected FE-chip are performed.

Within the YARR-software, the HW-controller and the FE-chips are completely decoupled. From a software perspective there are no specifications to the type of FE-chip connected to a certain HW-controller. The only requirement to the HW-controller are the availability of at most 32 Tx- and Rx-channels. The HW-controller sends certain commands to active Tx-channels and receives data from active Rx-channels without knowing the entity lying on the other end of those channels. Hence, the HW-controller along with the connected FE-chips can be specified during runtime via respective configuration files.

2.2.3 Enabling/ Disabling of Rx/Tx-channels

The YARR-software manages Tx- and Rx-channels in two different classes. Tx-channels are managed within the `TxCore` and Rx-channels within the `RxCore`. Every HW-controller needs to have an implementation of both those classes. Both include an object of type `uint32_t` called `enMask` (short for enable mask). As the name suggests, the integer masks the enabled and disabled channels. If a bit is set to 1, this is interpreted as an active Tx-/Rx-channel, if a bit is set to 0 the channel is disabled. This limits the number of connected Tx-/Rx-channels to 32. It should be noted that the `enMask` of a HW-controller's `TxCore` can be different from the `enMask` in the HW-controller's `RxCore`. Theoretically it would be possible to send data to certain FE-chips and receive data from others.

2.2.4 The scanConsole

All components that were just outlined converge in the central process of the YARR-software, the scanConsole. Primarily, the scanConsole processes the input given to the program at runtime. The required input includes the scantype, a HW-controller configuration file and a global FE-chip configuration file. The configuration files are encoded in the mark-up language JSON. Other possible command line parameters include the option to plot histograms or change the output directory. A command line issuing a digitalscan can look as follows:

```
$ bin/scanConsole  -s digitalscan  -r HW_config.json  -c FE_Config.json
```

After checking for all command line parameters, the configuration files are opened and the HW-controller as well as all FE-chips are configured according to the given parameters. When the configuration is completed, the scan is started. During the scan, a histogramming object collects the received hits. The hit data is then analyzed by the software. Finally, the scanConsole gives some information on the time needed by specific tasks to complete. A flow chart of the entire scanConsole can be seen in Figure 2.4. It should be noted that the histogrammers and the analysis run parallel to the scan. This results in an overall shorter runtime for single HW-controllers.

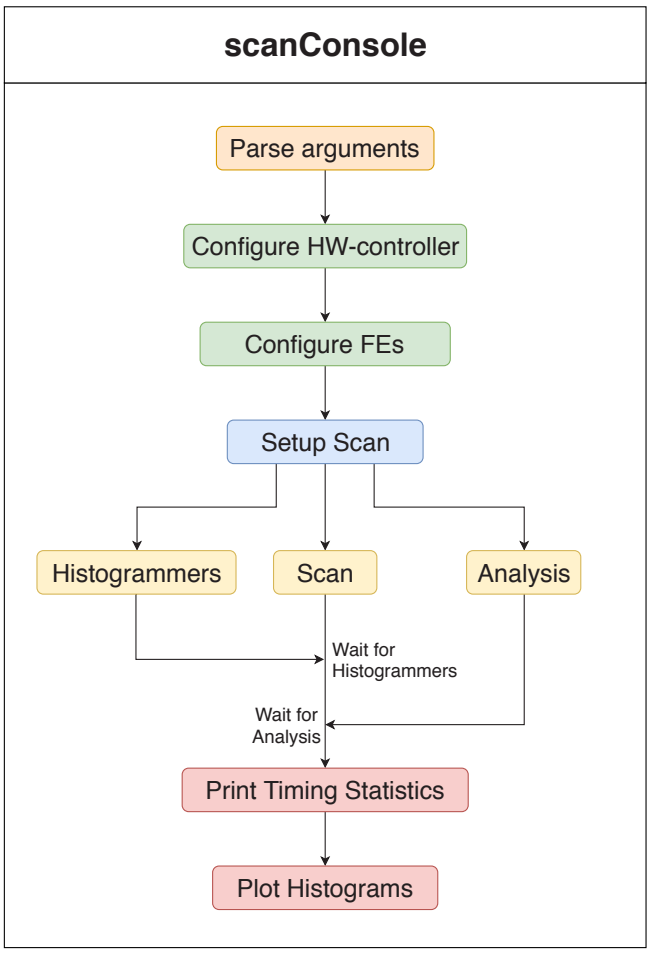


Figure 2.4: A schematic of the actions performed in the YARR scanConsole.

2.3 Multithreaded programming

In general, a program processes all of its commands sequentially in the order in which they were given. In some cases, however, it can be useful to have certain tasks be performed concurrently. The programming technique that allows this type of programming is called multithreading. Each thread consists of a sequence of instructions that are followed independently from the rest of the program. Different threads, however, can access shared resources of the program in which they are instantiated. The main method of any C++ program can be interpreted as one thread, the main thread. From here, other threads can be spawned [9].

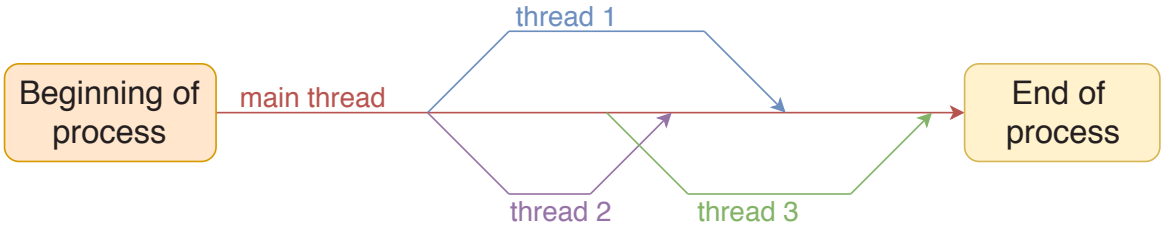


Figure 2.5: An illustration of the concept of multithreaded programming.

As Figure 2.5 suggests, all threads need to rejoin with the main thread after they have completed their tasks. If that does not happen and the main thread finishes its tasks before a concurrently

running thread, the program will crash.
Different threads do not have to actually run concurrently to each other. In cases in which the processor only has a single core concurrent processing is not possible. Instead each thread is consecutively given some processing time to proceed with its tasks. In a multi core CPU, however, it is possible to process different tasks on each core. Thus, real concurrence for threads is easily achievable.
There is a problem with multithreading that arises from the access to a shared resource by multiple threads. In some cases, particularly when the resource at hand can be written to, simultaneous writing of parallel threads is possible. This results in undefined behavior. If, for example, two threads write to the console simultaneously, the text on the console will be a scrambled output of both threads. Another problem arises, when two threads want to access the same variable. If both threads set the variable to a specific value, it is unclear what value this variable has after both threads have joined. This is because it is not specified, which thread sets the value of the variable last. The distribution of processing time among the threads happens, from a programmer's perspective, at random.
The latter problem has no specific solution. It is simply advisable to not access the same variable from separate threads. The former problem, however, can be solved. C++ introduces the class `mutex` (from 'mutual execution'). With such a `mutex` it is possible to manually lock the access to certain resource (e.g. `std::cout`). Thus, whenever a thread wants to access a shared resource, it blocks other threads from accessing it, until it no longer requires unshared access.
As multithreading can radically improve a process's performance, it can be of great use to the YARR-software. Some tasks, for example the histogramming of received hits, are already computed in separate threads. However, many tasks that could be performed in parallel are still performed consecutively within the YARR-software. On a high power PC it should generally be possible, to further increase the overall performance by implementing more threads into the software.

3 Coding implementation

In this bachelor thesis three features are added to the YARR-software. In Section 3.1 the limitation of the number of Tx- and Rx-channels is removed. After this, in Section 3.2, support for a variable number of HW-controllers is added. The HW-controllers perform their respective tasks simultaneously in different threads. To handle the parallel output of these threads, a novel logging class is implemented into the YARR-software. It is described in detail in Section 3.3. As a proof-of-concept the output generated by the FE-I4-digitalscan, the SPEC board and the emulator is reestablished using the logging class.

3.1 Support for any number of Tx/Rx-channels

As already pointed out, the current YARR-software sets the limitation on the number of connected Tx-/ Rx-channels to 32. This is not ideal, because for different HW-controllers the maximum number of connected FE-chips allowed by the hardware, can differ and even exceed 32. Thus, the way the YARR-software handles Tx- and Rx-channels is changed.
A reasonable approach to this problem is to diverge from the currently used method of masking enabled and disabled channels. Instead, it appears useful to store the channel numbers of all enabled channels in a data container with variable size. In C++ such data containers are best realized in the form of vectors. In this case a `std::vector` containing objects of type `uint32_t` is used. In Figure 3.1 the concept of the current management of Rx-/ Tx-channels is compared to the updated concept.

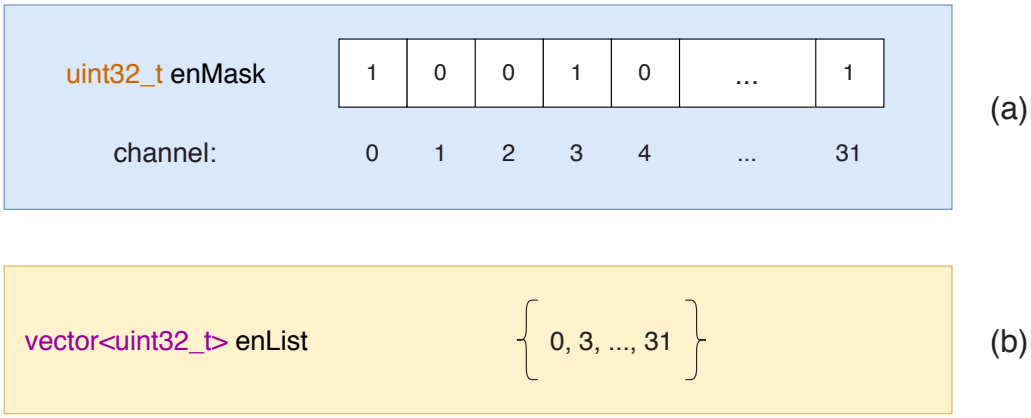


Figure 3.1: The current concept of Rx/ Tx-channel handling using a `uint32_t`-mask (a) compared to the updated concept using a `std::vector`, which stores the addresses of active channels.

This update entirely removes the software limitation on the number of connected FE-chips per HW-controller. The current YARR-firmware of implemented HW-controllers, however, still only supports 32 or fewer FE-chips. Thus, the HW-controllers' FPGAs still expect a 32-bit mask to enable and disable channels. Hence, the current YARR-system cannot yet make use

of this updated functionality. Instead, for the current system to work, the vector of active channels needs to be converted back into a 32-bit mask. This mask can then be sent to the FPGA. As an example, the `setCmdEnable` function in the TxCore, which makes use of this conversion, is presented. The function is required to change the active channels to a given set of channel addresses. These addresses are committed as a function argument. The code for the conversion from a vector to a 32-bit mask for the SPEC board's TxCore looks as follows.

```
void SpecTxCore::setCmdEnable(std::vector<uint32_t> vec) {
    uint32_t mask_tmp = 0; //integer mask to be sent to the SPEC board
    enList.clear(); //delete all currently active channels
    for(uint32_t num : vec){ //iterate over all channels to be enabled
        if (num > 31){
            ... //Error: Unknown channel address
        }
        else {
            mask_tmp |= (0x1 << num); //set bit of current channel to 1
            enList.push_back(num); //store current channel in
                                   //a member vector of the class
        }
    }
    SpecCom::writeSingle(TX_ADDR | TX_ENABLE, mask_tmp); //send mask to
                                                         //SPEC board
    ...
}
```

The function argument is a vector containing all channels to be enabled. First the list of active channels `enList`, which is stored internally in the `TxCore`, is cleared and a temporary `uint32_t` object named `mask_tmp` serving as a mask is created. It is followed by an iteration over all channel addresses contained in the committed vector. For every address the respective bit in the `mask_tmp` is set to 1 and the address is stored in the `enList`. Finally, the completed mask is sent to a specific address on the SPEC board via the function `writeSingle`.
This type of conversion between vectors and integer masks has been made for all implemented HW-controllers. Internally, however, the entire YARR-software now works with vectors containing active channels, rather than masks, when dealing with Tx-/ Rx-connectivity. Once the YARR-firmware has been adapted, only instances that directly communicate with the HW-controller will have to be changed accordingly.

3.2 Support for any number of Hardware controllers

The added support for more than 32 Tx-/ Rx-channels makes for more flexibility in the software regarding FE-chip-connectivity. Such flexibility for a varying number of HW-controllers is extremely beneficial for further development and usage of the YARR-software. Starting several processes to perform scans on multiple HW-controllers, as is necessary in the current state of the software, is rather complicated. Additionally, actions that are performed equally in both processes need to be performed separately from each other, thus wasting CPU resources and increasing the overall computing time. This problem is best solved by adding the support of multiple HW-controllers in one process. The actions specific to respective controllers can be performed in parallel threads. In this concept the main-thread in the scanConsole serves as a monitor for all HW-controller threads. In order to not have a scrambled output from all HW-controller threads, all output will be directed to an additional thread, whose sole purpose is the

output handling. The output concept will be outlined in detail at a later point in this thesis. A graph of the concept for the support of multiple HW-controllers can be seen in Figure 3.2.

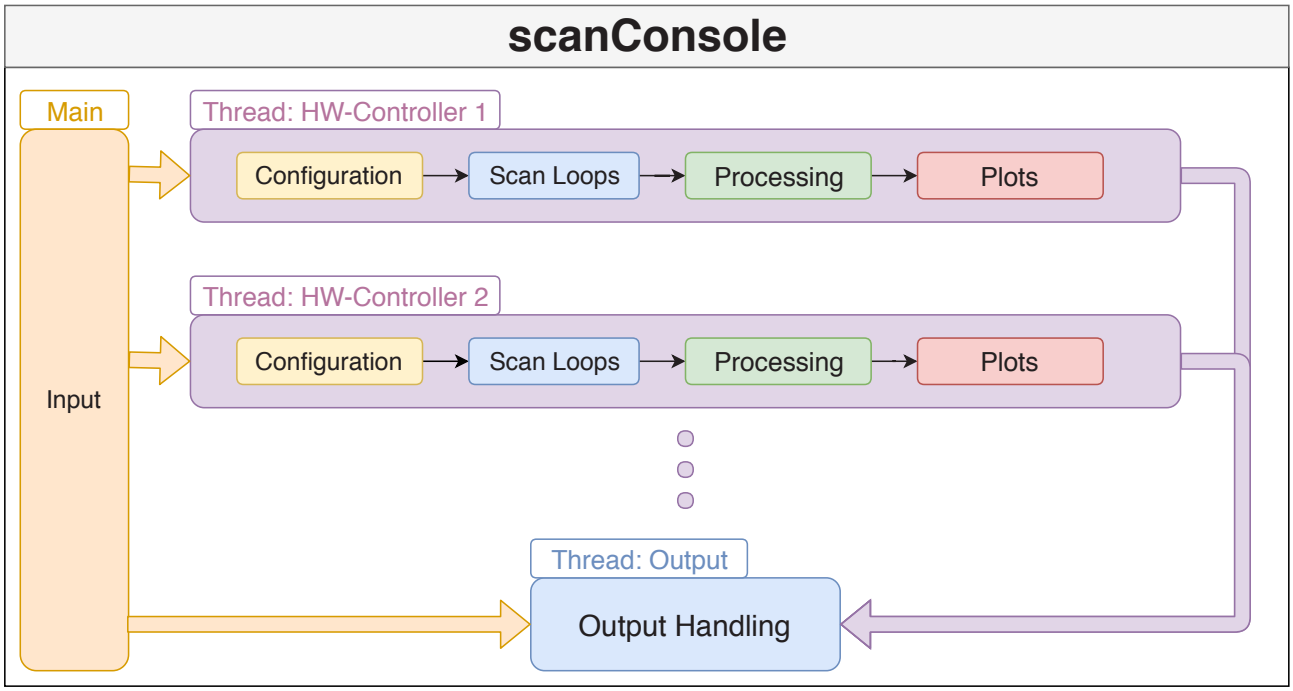


Figure 3.2: The used concept for the support of multiple HW-controllers in one scanConsole.

The main-thread processes the input, which is given at runtime. Then all HW-controller threads are started together with a separate output thread. The HW-controller specific actions like configuration, scanning, processing and plotting the received data are done in the HW-controller thread. The output from all threads is sent to the output thread, where it is stored and exported in a reasonable manner.
The general approach to implement this concept is to outsource all HW-controller specific actions from the main method of the scanConsole to a separate function. This function requires certain parameters for the actions to be performed, including the HW-controller configuration and the global FE-chip configuration file. If the main thread needs to access certain objects after they have been processed in the new function, only pointers to these objects are passed. Once the function has been implemented, it is generally possible to call it multiple times and in multiple threads. There is, however, no software support yet for defining multiple sources for HW-controller configurations. Moreover, there needs to be a way to assign each HW-controller a specific global FE-chip configuration file. The solution to this is to implement a global multi-HW-controller configuration file. In this configuration file any number of HW-controllers can be specified. A command line issuing a scan with multiple HW-controllers looks as follows.

```
$ bin/scanConsole  -s digitalscan  -a Multi_HW_config.json
```

Each HW-controller specified in the multi-HW-controller configuration file requires exactly one associated global FE-chip configuration file and a unique identification string. There cannot be any duplicate HW-controllers defined in the global configuration, because accessing a HW-controller simultaneously from two separate threads would lead to undefined behavior. The identification string is important, because it allows for a distinction between the output files of

different HW-controllers. The string is added to every filename as a prefix. Thus, it is easily possible to provide two different HW-controllers with the same FE-chip configuration and produce output configuration files with distinct names.
All paths to HW-controller configuration files and the paths to their respective FE-chip configuration files are stored in vectors to assure that there is no software limitation to the number of possible HW-controllers. During the storing process, the HW-controller configuration files and their identification strings are checked for duplicates. If duplicates are found, the program throws an exception. The detailed course of events in the scanConsole's main thread is depicted in Figure 3.3.

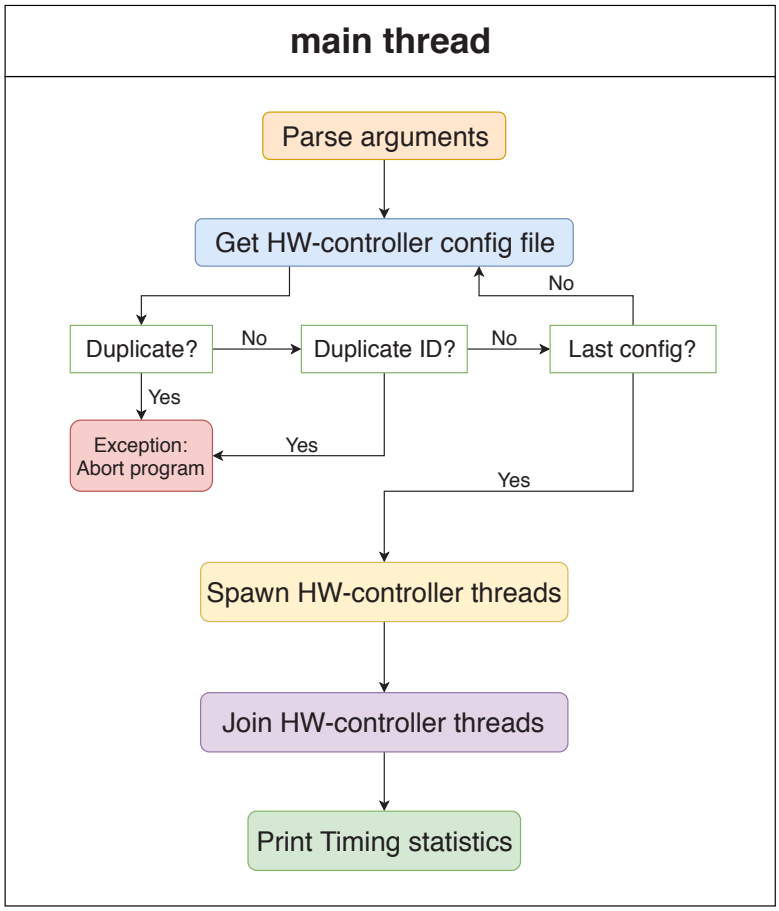


Figure 3.3: The concept of the main thread in the updated scanConsole.

Through these basic changes in the software, operations on multiple HW-controllers in one scanConsole should be possible. In order to be able to determine, whether there is a gain in performance that can be attributed to multithreading, an additional scanConsole is implemented into the program. This scanConsole does not rely on multithreading and instead performs the scans on all HW-controller successively. In all other aspects it resembles the scanConsole that was explained in this section.

3.3 Logging class

As has been mentioned in the previous section, the addition of parallel scans on multiple HW-controllers requires a new way of handling the output of the YARR-software. Currently, all output statements are issued using either `std::cout` or `std::cerr`, the standard C++ output streams. This is not ideal for several reasons. First of all, due to the support of multiple HW-controllers in parallel threads, the output of those threads is scrambled, when handled via `std::cout`. It would be possible to lock access to the `std::cout`-resource via a `mutex`, but simply locking access to a resource that is accessed frequently will slow down the entire program. If certain threads need to print output, but the output resource is blocked, they simply have to pause their activity until the resource is available again. Thus, any approach using mutexes frequently is not usable in practice.
Such an approach also doesn't solve the second problem arising from using `std::cout`: All output is of equivalent status and cannot be turned off separately. This, however, is very important. Not all debug output needs to be shown when using the software in operation and sometimes it can be useful to disable the output entirely (e.g. to save processing time). The software itself should know, what output is most urgent and what output is simply there for the purpose of having a better overview over the current state of the process.
The only reasonable solution to these problems is the implementation of a logging class. Such a logging class would handle all output in the YARR-software. It needs to fulfill certain requirements, in particular be thread safe and have multiple verbosity levels. There are a number of already existing logging classes that meet those requirements. In this thesis, however, a custom logging class for the YARR-software is conceptualized and implemented into parts of the software. The advantage of such a logging class is that certain additional requirements unique to the YARR-software can easily be implemented.
The idea behind this logging class, in the following called `Logger`, is to give every thread an object of type `Logger`. This includes the main thread. When constructed, this `Logger` is assigned a certain verbosity level. All output messages will be given a certain verbosity level as well. They are added to a container within the `Logger` of their respective thread, instead of being printed to the console right away. However, before being added to the container, the verbosity level of the message at hand is compared to the verbosity level of the respective `Logger`. Should the message's verbosity exceed that of the `Logger`, the message is ignored. A parallel running output thread (see Figure 3.2) has access to all `Logger` objects. Inside this thread, the messages stored in their respective `Logger` container are printed in a readable and reasonable way. The concept is visualized in Figure 3.4.

3.3.1 Components of the Logging class

The verbosity levels of the `Logger` in ascending order are as follows: FATAL, ERROR, WARNING, INFO and DEBUG. The level that most accurately resembles the verbosity of the current YARR version is INFO. Thus it is used as the `Logger`'s standard verbosity level. Every type of message has a predefined prefix, depending on its verbosity. For example, the prefix for messages of verbosity ERROR have the prefix "Error: ". This unification of message types makes the entire YARR-software more readable and consistent.
The container in which messages are stored is chosen to be a `std::queue`. A `queue` has the advantage, that it is filled from one side and emptied from the other. Thus it allows the order of the messages to be easily maintained while printing.
Currently, there are two functions for printing messages. The first function `void printOne()`

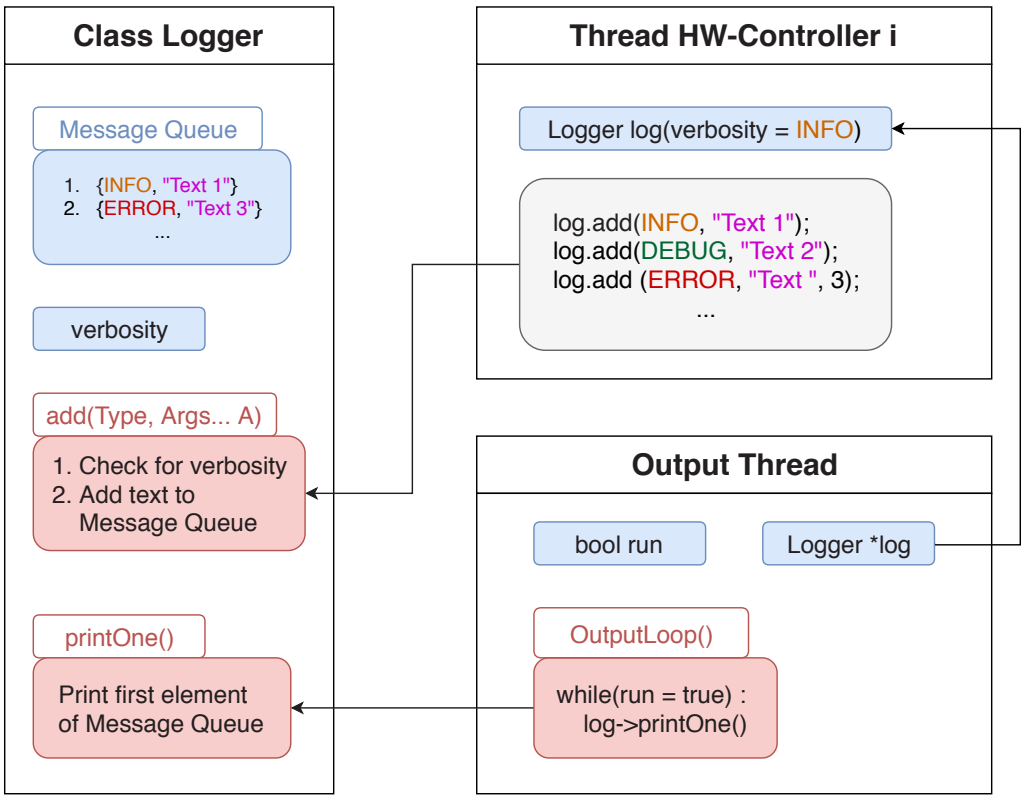


Figure 3.4: The concept of the custom logging class for the YARR-software.

removes the first message from the message `queue` and prints it. Messages of verbosity INFO and DEBUG are printed to the `std::cout` stream, messages of verbosity FATAL, ERROR and WARNING to the `std::cerr` stream. The second function `void printAll()` does the same thing for all current elements of the message `queue`.
Messages are added to the `queue` via the `add` function. The function requires first the verbosity of the message and then the message itself as arguments. To make the function as flexible as possible, it is implemented as a variadic function template. Such a template allows for an indefinite number of function arguments of varying types. Thus, it is possible to add segmented messages of any length, which makes the use and the implementation of the `Logger` more convenient. The code for this template looks as follows:

```
template<typename... Args>
void add(Type_Flags type, Args... args){ //any number of arguments, first argument
                                         //is message's verbosity
    if(type <= verbosity) { //check message's verbosity
        ... //store verbosity

        std::stringstream ss; //create stringstream to temporary hold the message
        to_sstream(&ss, args...); //add arguments to the stringstream

        ... //add string contained in ss to the message queue
    }
}
```

The function `to_sstream(std::stringstream *, Args... args)` is another function template, which iterates recursively through all arguments given and adds them to the `stringstream`.

The use of this `stringstream` allows for more flexibility in the types of the given arguments. Basically all output messages can keep their format. As an example the command

```
std::cout << "The value given is " << std::hex << std::showbase << value << std::dec << std::endl;
```

would be changed to

```
log.add(INFO, "The value given is ", std::hex, std::showbase, value, std::dec);
```

if `log` was the name of the `Logger` in this thread. It is not necessary to include the `std::endl` command, because the `Logger` adds this command to every message it prints. The object `value` is of type `int` in this example. Every type compatible with the `<<` operator in `stringstreams` is a possible argument of the `add` function.
The last feature that the `Logger` in its current state has is the option to send output to a file. Even if the output of different threads is not scrambled, it can be difficult to distinguish the output of those threads in one console. As a primary incentive to overcome this problem, each logger is provided with a background color. All messages that this logger sends to the terminal are printed with that background color. This, however, is not a general solution to the problem, as this type of output can be rather confusing, especially if too many HW-controllers are scanned parallel. Thus, the `Logger` can be provided with an optional path to a .txt file, to which all of its output will be printed. Messages of verbosity FATAL and ERROR will additionally be printed to the console with the background color of the respective `Logger`.

3.3.2 The Output thread

The output thread is a separate entity from the `Logger`, but it is crucial to the overall functionality of the logging concept presented here. Its task is to print the messages of all `Loggers` in a way that is readable and functional. There are basically two approaches to this. Either the output thread waits for a certain different thread (e.g. a HW-controller thread) to finish its tasks and then prints all of its output consecutively. This would make the output much more readable, however, the entire output would be delayed by an unknown amount of time. The latter effect makes this approach undesirable, because in the case of an error or a warning a real time output is required.
The second approach makes for a less sorted, but immediate output. The output thread simply loops over all `Loggers` until a certain signal from the main thread is given. In the context of this thesis, that signal means that all HW-controller threads have joined. Inside the loop either of the functions `printOne()` or `printAll()` is called for all HW-controllers. This ensures, that all output from those controllers is printed as fast as possible. In this program the output thread was chosen to call the `printAll()` function for all `Loggers`.

4 Performance and Functionality tests

In this chapter the functionality of the added components is tested and their implications on the overall performance of the YARR software is determined. The tests are performed on a high power PC with a four core CPU and 32 GB of RAM. The PC is connected to an XPressK7 FPGA board [10] and a KC705 FPGA board [11], which both have the YARR SPEC-firmware installed. Both boards are connected to one IBL half-stave with 16 mounted FE-chips respectively. A schematic of an IBL stave can be seen in Figure 4.1. Only certain FE-chips of those are functional, because the stave has been in use for several years and was damaged in the production process. The tests performed in this thesis, however, do not rely on a large number of available FE-chips.

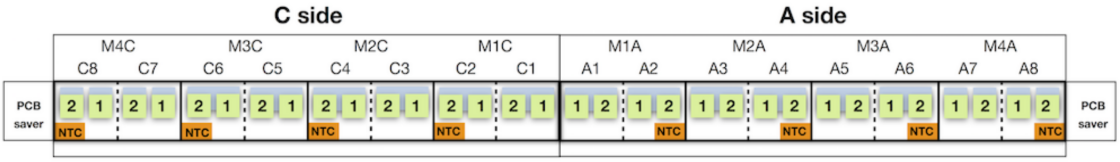
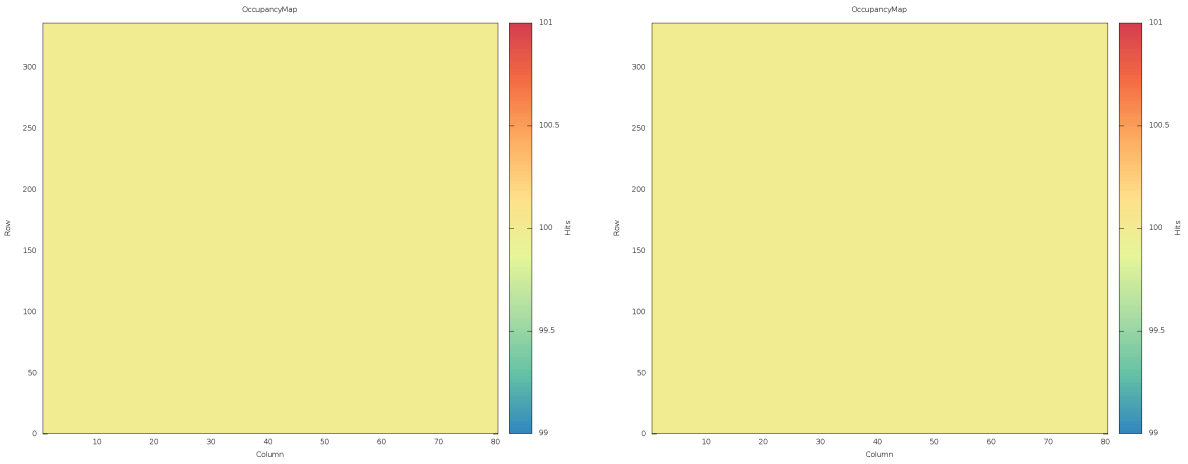


Figure 4.1: The schematic of an IBL stave. The entire stave is divided into two half-staves (A- and C-side). The different modules are marked as C1-C8 and A1-A8. Every module is connected to two FE-chips [8].

More specifically, every performance test performed is structured in the following manner. One test run consists of 2,000 consecutively performed digitalscans. For each HW-controller to be scanned, only one FE-chip is addressed. For the XPressK7 board the addressed FE-chip has Tx-channel 8 and Rx-channel 8 (chip name M2C4-2) and for the KC705 board the addressed FE-chip has Tx-channel 8 and Rx-channel 9 (chip name M3A5-2). For better comparability, all scans of the respective HW-controllers are only performed on those chips. As a third HW-controller, the YARR emulator is used. It also performs the scans on only one emulated FE-chip. The different test runs differ in the number and the type of connected HW-controllers and the version of the scanConsole. In total there are three versions of the scanConsole used in the tests: the modified version that uses multithreading for scans on different HW-controllers, the modified version that scans different HW-controllers successively and the original scanConsole of the unmodified YARR version.
The performance of the software is measured as the runtime of the digitalscans. This runtime is determined within the YARR software. The point in time at which the program is started and the point in time at which the scans are completed are determined and compared. The program then displays the runtime in milliseconds.
In the following, every implementation discussed in Chapter 3 will be tested separately. In Section 4.1 the functionality of the YARR-software after the added support for more than 32 Tx/Rx-channels is tested. In Section 4.2 the performance of scans with several HW-controllers is determined and the implications of multithreading on the overall results is discussed. Finally, in Section 4.3, the impact of the logging class and output in general on the performance of the YARR-software is evaluated.

4.1 Functionality of updated Tx/Rx-channel handling

The first addition to the YARR-software in this thesis was the support for more than 32 connected FE-chips per HW-controller. However, the available test setup only consists of HW-controllers with 16 FE-chips connected to each one. Thus, it is only possible to test the general functionality of the software with the changed Tx/Rx-channel handling. In addition to this, one can check for possible influences on the performance of the software.
To test the general functionality, the results of digitalscans before and after the modification are compared. Figure 4.2 shows the two hitmaps, which were acquired through scans on the FE-chip M2C4-2, the standard testing FE-chip in this thesis. The digitalscan triggered every pixel 100 times.



(a) Hitmap with original YARR-software

(b) Hitmap after changed Tx/Rx-handling

Figure 4.2: Comparision between the hitmaps of a digitalscan on the FE-chip M2C4-2 with different versions of the YARR-software. The color indicates the number of hits generated in a specific pixel.

The results show that the changed internal management of Tx/Rx-channels does not change the scan results of the FE-chip. In both scans every pixel reported 100 hits. This does not only show that the software works, but also that all pixels of the FE have a functioning connection to the readout system. Similar results could be achieved with up to eight FE-chips in one scan. More functioning FE-chips were not available on one half-stave.
The addition to the software does not impact the performance negatively at all. The computing time with the changed software averages 2.15(2) s. This is exactly equal to the computing time of the original software of 2.15(2) s. One might have assumed that the conversion between `uint32_t` and `std::vector` negatively affects the overall computing time. Such concerns have now been proven unjustified.
In conclusion, the added support for more than 32 connected FE-chips per HW-controller is exclusively beneficial. It is entirely functional and allows for more flexibility in the setup of a YARR readout system. The added computational effort that arises from the use of a `std::vector` does not have any consequences for the overall computing time.

4.2 Functionality and Performance tests for multiple Hardware controllers

After the added support for multiple FE-chips per HW-controller has been shown to not affect the overall performance, now the impact on the performance by the added support for multiple HW-controllers is determined.
At first, the functionality of the software addition is tested. After this, the software undergoes a number of elaborate performance tests to ascertain in how far the added software components, in particular the use of multithreading, affect the computing time. In the course of these tests, the distribution of the computing time of digitalscans is examined in more detail.
As already mentioned, there are three types of scanConsoles, on which tests can be performed. The first one is the scanConsole of the modified software, which uses multithreading to scan different HW-controllers at the same time. The second scanConsole also uses the modified software, however, it scans different HW-controllers successively. The third and last scanConsole is the original scanConsole of the unmodified YARR-software. It does not support scans on multiple HW-controllers in one instance.
The functionality of the software is tested by changing configuration settings and deliberately causing exceptions. The different tests show that it is possible to perform scans on any number and combination of emulators and SPEC boards. The catching of exceptions, like a duplicate HW-controller configuration file or syntax errors, works reliably. Furthermore, all tools and features of the original YARR-software are compatible with the modified software. All the test results show that the modified software functions properly. Thus, it is legitimate to interpret the results of the performance tests as representative for the performance of the software in actual application in the ITk-readout system.

4.2.1 Distribution of the runtime

As aforementioned, every test run consists of 2000 consecutively performed digitalscans. Among the different tests in this section the number and types of HW-controllers are changed. For every test the average runtime of all 2000 scans is taken as the measured value of the actual runtime. The square root of the variance of the data set is interpreted as the error. The actual distribution of the program's runtime is not important for further analysis. It is, however, reasonable to consider the distribution in order to check for any irregularities. Additionally, any correlation between the runtime of consecutive scans needs to be factored into all further analysis. The distribution of the runtime of scans on one SPEC board with the multithreaded scanConsole is taken as example. A histogram of all the measured runtimes can be seen in Figure 4.3.

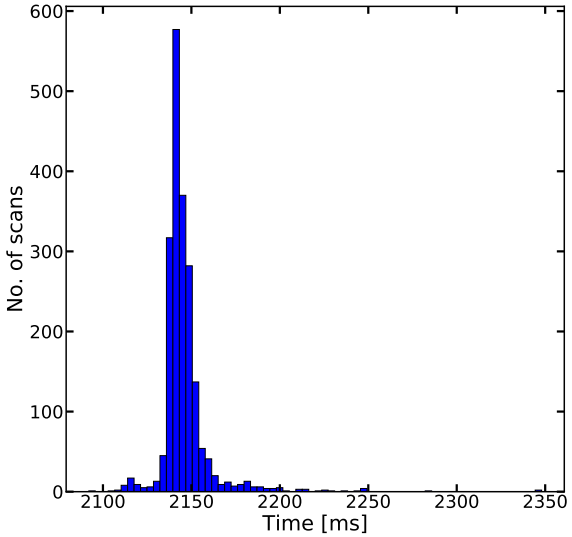
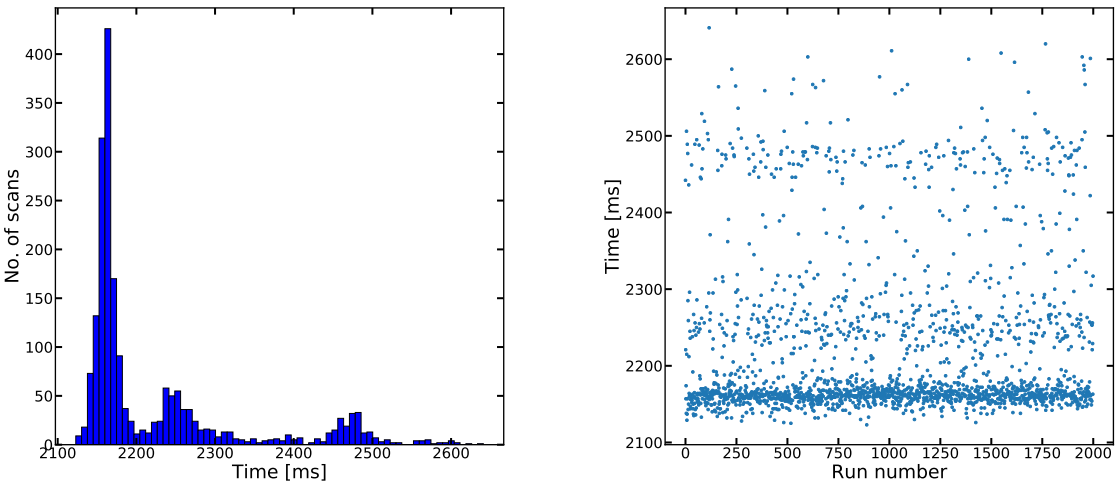


Figure 4.3: Histogram of measured runtimes of one SPEC board scanned with the multithreaded scanConsole.

The distribution shows a very narrow peak at 2146(15) ms. With very few exceptions, all measured values are in this peak. The shape of the distribution might be approximated by several density functions, like a Gaussian or a gamma distribution. There is, however, no need for an analytical description of the runtime distribution, as only the average and the variation are used for further analysis. Thus, a fit of such distributions to the density is forgone. Most of the acquired data sets follow a similar distribution. Nonetheless, there are some exceptions to this.



(a) Histogram of runtimes

(b) Runtimes plotted against run number

Figure 4.4: Histogram (a) and runtime plotted against the run number (b) for scans performed on two SPEC boards with the multithreaded scanConsole.

A scan on two SPEC boards with the multithreaded scanConsole produces the distribution visible in Figure 4.4 (a). There are some obvious differences to the distribution in Figure 4.3. In addition to the main maximum, there are two less distinct peaks. These are at a runtime which is higher than the runtime of the main peak. This may be due to background processes running on the PC, which was used for testing. To check, whether the peaks arise from random processes, the measured runtimes are plotted against the respective run number in Figure 4.4 (b). There does not seem to be any correlation between the run number and the runtime. The correlation coefficient of 0.04 between the two quantities confirms this. This indicates that the lengthened runtimes do not arise from random processes in the background, but instead have some more fundamental cause. Possible candidates to explain this phenomenon are the analysis and processing threads, which run in parallel to each HW-controller thread. They require much fewer resources and their tasks are completed quicker than the main HW-controller thread. However, for multiple HW-controller threads they can cause minor shifts in the runtime. The reason for this is that there are only four CPU cores available in the test setup. A thread can only run on one CPU core, which needs to be specified at the start of the thread. The CPU distributes the threads to the four cores in the order in which they are started. Since the order in which the processing and analysis threads of different HW-controllers are started can vary, these threads are distributed in a quasi random manner among the four CPU cores. It can now happen that two very time-consuming threads are assigned to the same CPU core. This would lengthen the overall runtime compared to the case in which rather time consuming threads are paired with smaller threads on the same core. The resulting runtime distribution would resemble the one in Figure 4.4 (a). Such distributions can be observed for all test runs performed with multiple HW-controllers on the multithreaded scanConsole, especially for test runs with emulators, since those have additional threads for generating data.
This phenomenon results in an increasing variance in the runtime for higher numbers of scanned HW-controllers. The validity of using the variance as an error for the expected runtime, however, is not impacted by this observation. Nevertheless it is important to have identified the cause for this distribution, as it might explain other findings in the following analysis.

4.2.2 Runtime analysis

The main question to be answered in this section is: Does scanning different HW-controllers in separate threads reap a benefit for the overall runtime and if so how much? For this purpose, several configurations of HW-controllers are scanned and the runtime is measured. Scans are performed on the following combinations of HW-controllers:

- 1 - 2 SPEC boards
- 2 SPEC boards and 1 emulator
- 1 - 7 emulators

All of these combinations are tested on the multithreaded scanConsole as well as on the scanConsole that scans different HW-controllers successively. To be able to compare the results to the unmodified YARR-software, scans on each of the SPEC boards and on the emulator are performed with the original scanConsole. For each configuration, the runtimes of the individual HW-controllers are added up. This way of comparing the runtimes is legitimate, as scanning all HW-controllers consecutively in separate processes would be the appropriate way to scan multiple HW-controllers with the unmodified software.
The resulting runtimes for all configurations can be found in Table 4.1. For further analysis, the configurations that feature SPEC boards are examined separately from the configurations including exclusively emulators. The reason for this is that the emulator generates all the data it receives in an additional thread, which requires much processing power. Especially with regard to testing the benefits of multithreading, runtimes of scans featuring SPEC boards and runtimes of scans on emulators are incomparable.

Table 4.1: Resulting runtimes from tests regarding multiple HW-controllers. With the modified software, scans are performed in parallel (multithreaded) and successively for multiple HW-controllers. With the unmodified software, only a successive scanning of multiple HW-controllers is possible.

Test Setup	Runtime [s]		
	Modified scanConsole		Original scanConsole
	Multithreaded	Successive	
1 SPEC	2.15(2)	2.15(2)	2.15(2)
2 SPECs	2.22(11)	4.28(2)	4.29(2)
2 SPECs + 1 Emulator	2.44(28)	6.97(8)	7.00(5)
1 Emulator	2.71(8)	2.69(5)	2.71(4)
2 Emulators	3.37(8)	5.39(7)	5.42(8)
3 Emulators	4.82(10)	8.01(14)	8.13(12)
4 Emulators	8.05(24)	10.72(20)	10.84(16)
5 Emulators	11.89(28)	13.42(20)	13.55(20)
6 Emulators	15.58(29)	16.11(22)	16.26(24)
7 Emulators	20.4(7)	18.83(29)	18.97(28)

The runtimes of all tests performed on configurations featuring a SPEC board can be seen in Figure 4.5. The different runtimes are marked according to the type of scanConsole used.

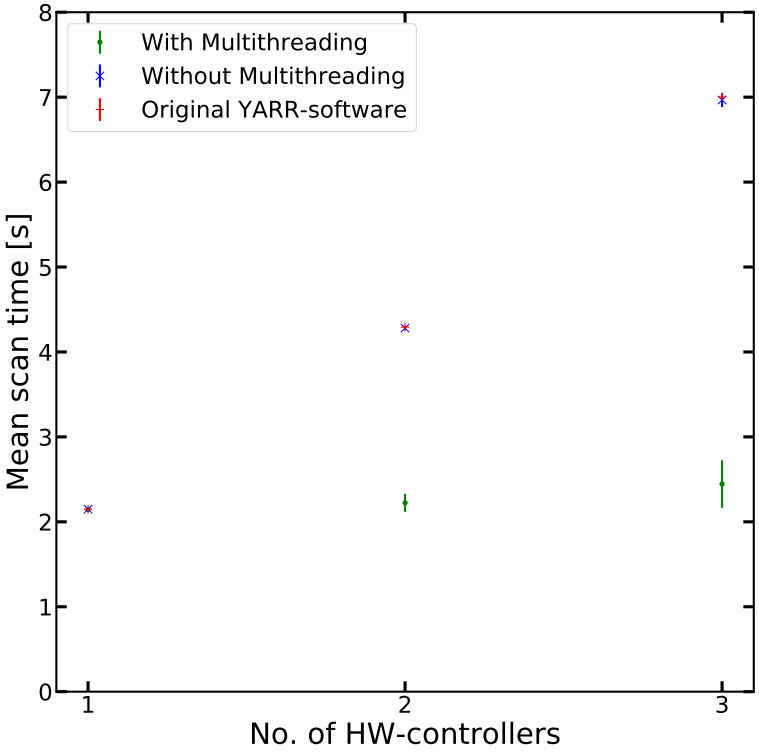


Figure 4.5: The runtimes measured for test configurations featuring the SPEC board on varying scanConsoles.

Figure 4.5 clearly shows an enormous reduction in runtime for the multithreaded scanConsole. For one SPEC board, scans with all three scanConsoles require 2.15(2) s. This shows that the changed initialization of HW-controllers does not negatively impact the overall runtime.
For the tests with multiple SPEC boards there is a clear difference between the multithreaded scanConsole and the other two. The modified scanConsole with consecutive scanning of HW-controllers and the original scanConsole do not differ much in terms of their runtime. For each HW-controller the runtime of those scanConsoles increases by about 2 - 2.5 s. The runtime of the multithreaded scanConsole, however, does not increase much. For two SPEC boards there is no big difference to the runtime with just one SPEC board. With the emulator as a third HW-controller, the runtime goes up to 2.44(28) s. This corresponds to the runtime of 2.71(8) s of a scan on just one emulator. This suggests that the data generating of the emulator is mainly responsible for the longer runtime with three HW-controllers. Regardless, this measurement shows that for SPEC boards there is a clear gain in performance through multithreading. More precisely, the use of multithreading reduces the runtime of scans on two SPEC boards by 48(3),% and for scans with two SPEC boards and one emulator by 65(4) % compared to the original scanConsole. It is, however, to be expected that this decrease in runtime does not continue indefinitely. The PC used for testing has four CPU cores. This means, for five or more HW-controllers processing time of some of those cores needs to be distributed among the HW-controller threads. Thus the relative saved time should no longer increase with the number of HW-controllers. In fact, it is likely that the saved time will at some point decrease with the number of HW-controllers. This is, because the creation of threads and their management

require processing power. The required processing power should increase with the number of parallel threads. The following paragraph shows this tendency by means of the tests performed exclusively with emulators.
For the tests conducted with configurations consisting purely of emulators, the runtimes for the different scanConsoles are visualized in Figure 4.6.

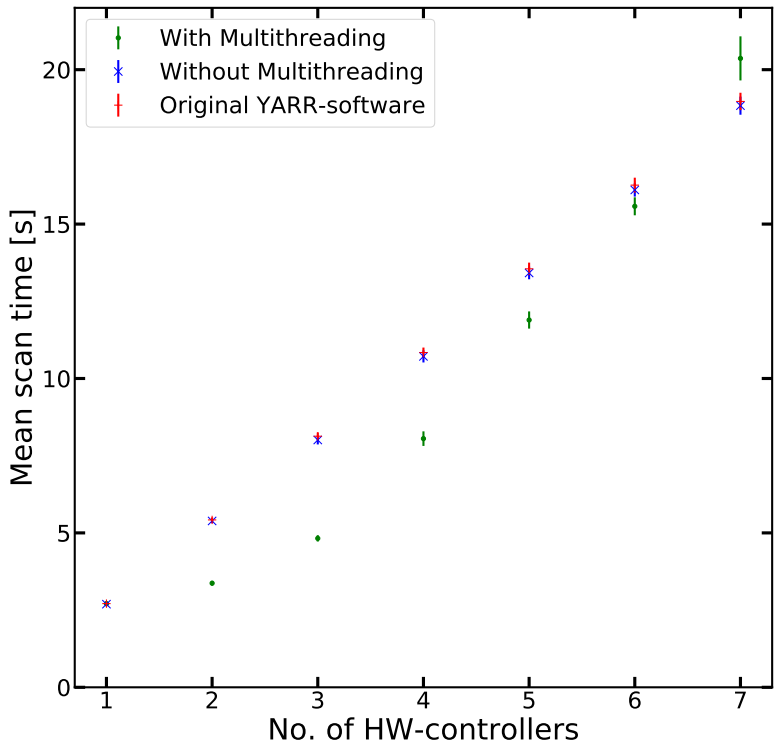


Figure 4.6: The runtimes measured for test configurations featuring only emulators on varying scanConsoles.

Primarily, it should be noted that up to a number of five emulators there is a significant amount of time saved through multithreading. There are, however, some obvious differences to Figure 4.5 visible. For instance, the runtime is generally longer than for the scans on SPEC boards. The most prominent difference is the strong increase of computing time for the multithreaded scanConsole. For seven emulators the runtime with the multithreaded scanConsole even exceeds this of the non-multithreaded and the original scanConsole. This is a significant result. It means that multithreading is only beneficial to the overall performance of the YARR-software, if the number of HW-controllers stays low enough. The reason for this has already been mentioned. For every thread created in a program, a certain amount of processing time is required to assign the thread to a core and start its processing. Furthermore, there needs to be some internal thread-management in the CPU, which schedules the processing time of all threads. This management requires more resources, the more threads are active. For emulators the number of active threads and the complexity of those threads is much larger than for controlling FPGA boards. Every emulator has a general HW-controller thread which initializes the scan, the histogrammers and the analysis. The scan, the histogrammers and the analysis are all

performed in separate threads. For the emulator there is an additional data generating thread running in parallel to all those threads mentioned. Multiply this by the number of scanned emulators and add the main- and the output-thread and it becomes clear, why there is no time saved by scanning too many emulators at the same time.
These results can be evaluated even further. In Figure 4.7 the relative saved time for the tests with SPEC boards and with emulators is plotted against the number of HW-controllers.

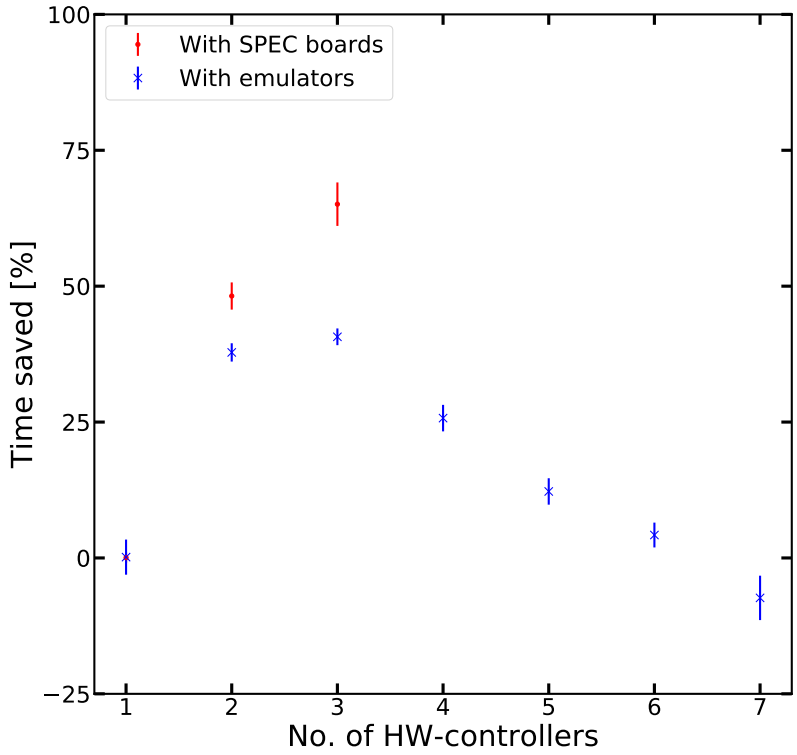


Figure 4.7: The relative saved computing time through multithreading compared to the original scanConsole.

It is clearly visible that the relative saved time for scans exclusively with emulators is generally lower than for scans with SPEC boards. The biggest difference can be observed between scans with three HW-controllers. The relative saved time for the scans with SPEC boards is 65(4) %, while for the scans without SPEC boards it is only 41(2) %. Already at numbers of HW-controllers this low the higher number of threads for emulator scans has measurable consequences. Coincidentally, three emulators is also the number of emulators for which the relative saved time through multithreading reaches its maximum. For higher numbers of emulators the relative saved time decreases linearly and for 7 emulators there is no more time saved through multithreading at all. The linearity of the decline shows that every added emulator deteriorates the performance to the same degree. This coincides with the proposed explanation for the decrease. The creation and management of all emulator-specific threads should take up the same amount of resources for every added HW-controller. Thus, once a maximum has been reached and all cores are used to capacity, every added HW-controller should impair the performance in equal measure.

It can be expected that such a decrease in relative saved time can also be observed for a higher number of scanned SPEC boards. The decrease for those, however, should be much slower and start at a higher number of scanned SPEC boards. This is to be expected, because SPEC boards do not have as many active threads during a scan as emulators. The maximum relative saved time in this setup, from a hardware perspective, is about 75 %. This is due to the used PC having four CPU cores. Whether this maximum can be reached cannot be determined, because there are only two SPEC boards available for testing. In general, for every scan setup an optimal number of HW-controllers can be found, for which the relative saved time through multithreading is maximized.
In conclusion, the distribution of computation to multiple threads reaps great benefits for the overall computation time. With three independent HW-controllers a runtime reduction of 65(4) % could be achieved compared to the unmodified YARR-software. This makes this feature ideal for application in the ITk-readout system. It should, however, be noted that there is a maximum number of HW-controllers, which, when exceeded, causes the multithreaded scanConsole to run slower than its successively scanning counterparts. There is also an ideal number of HW-controllers, at which the relative saved time through multithreading is maximized.

4.3 Influence on Performance by the Logging class

The last feature that requires testing is the Logging class. Again the feature's functionality and its performance impact are determined. The addition of different verbosity levels, however, also allows for testing the general impact of output on the YARR-software. By changing every logger's verbosity levels to ERROR, only error messages are displayed on the console. All other output is completely ignored by the loggers. This is almost equivalent to having no output at all. The only difference is that there is one `if`-statement for every message, which checks for the message's verbosity. This statement, however, has almost no relevance regarding the runtime compared to the actual output to the console. Thus, this test run should indicate, whether the output of the YARR-software has any serious implications on the overall performance.
In another test run all HW-controller-specific output is redirected to logging files. This feature of the logging class might influence the performance. It is not clear how sending output to files compares to sending the output to the console. All tests discussed in this section are performed on a SPEC board. In total, performance tests with the following settings are performed:

- General scan (with default verbosity level)
- Scan without output (verbosity ERROR)
- Scan with output to logging file

The functionality tests show that the functionality of the logging class is just as reliable as the other two added features. The output of different threads is no longer scrambled and it is possible to differentiate between the output from different HW-controllers. The results of the performance tests are listed in Table 4.2.

Table 4.2: Resulting runtimes for tests regarding the logging class.

Test Setup	Runtime [s]	
	Modified scanConsole	Original scanConsole
Without Logger	-	2.15(2)
With Logger	2.15(2)	-
No output	2.15(1)	-
To file	2.15(1)	-

As the results show, the scan time of scans using the logging class does not differ from the computing time of the unmodified YARR-software. Furthermore, neither redirecting nor disabling the output changes the average computing time significantly. This comes as a surprise, as output is generally known to be a strong impairment for a program's performance. The scan itself and the data processing seem to have a much greater impact on the performance than the output does. It is probable that for longer scans or for multiple, consecutive scans there is an increase in performance through deactivating the output. More elaborate tests on the impact of output can be made, when the logging class is implemented into the entire software.
In conclusion, the logging class functions exactly the way it is supposed to. It allows for unscrambled, parallel output without impairing the overall performance. For more clarity and a better overview, the entire output can also be streamed to separate files, which does not influence the performance either. Generally, it has been found that output has no significant impact on the overall computing time of scans with the YARR-software.

5 Summary and outlook

In this thesis three features were added to the ITk-DAQ-software. The first of these features is the possibility to connect more than 32 FE-chips to one HW-controller. This limit had been present, because the active Tx/Rx-channels to different FE-chips were specified in the software via a 32-bit mask. Every bit in this mask represented one Tx/Rx-channel, with a value of 1 meaning a channel is active and 0 meaning a channel is disabled. In this thesis, this concept has been replaced by storing the addresses of active channels in an `std::vector`, whose size is not limited in any way. From the software side it is now possible to add an indefinite number of FE-chips to a HW-controller. The firmware on all supported HW-controllers of the YARR-software, however, has not yet been adapted to this new feature and still only supports 32 FE-chips or fewer. Performance tests showed that the runtime of scans is not influenced by this modification.
The second addition to the software is the possibility to perform scans on multiple HW-controllers in one scanConsole. These scans are performed in separate threads with the intention to reduce computing time. Tests on a high power PC with a four core CPU showed that there is a clear reduction in runtime due to multithreading. This reduction goes as high as 65(4) % for two SPEC boards and one emulator as parallel scanned HW-controllers. The reduction in time for emulators turns out to be smaller, because of the additional threads needed to generate the scan data. A maximum in the relative saved time through multithreading can be observed for scans with emulators. It can be explained by the processing power needed to create and manage a high number of parallel threads. Such a maximum can be expected for all types of HW-controllers and all PCs. It constitutes the ideal number of HW-controllers to be scanned in parallel by a specific PC.
The last feature added to the YARR-software in this thesis is a custom logging class. It was a necessary addition, because the output created in parallel threads is scrambled when streamed via `std::cout`. The added logging class prints output from parallel threads to the console line by line and in different colors. It is also possible to stream specific output to a logging file, which makes for an even more readable output. Additionally, the verbosity level of every logger can be adapted for different purposes. This allows for testing the general impact of output in the YARR-software. Test results show that there is virtually no influence to the overall performance by output messages.
In summary, all additions to the YARR-software make for much more flexibility in the connectivity of hardware components. The logging class also allows for more control over the output of the software. On the basis of these new additions, further development can be initiated.

One way in which the added features can be improved regards the support for multiple HW-controllers. Currently, it is only possible to perform scans on HW-controllers that are connected to the PC executing the scanConsole. Since one PC can only handle a limited number of connected HW-controllers, it would be beneficial to be able to connect to HW-controllers on different PCs. The most sensible concept that could realize this would need one monitoring process issuing scanConsoles on different PCs. This monitoring process should also receive all generated output and data from the different scanConsoles. An illustration of such a concept

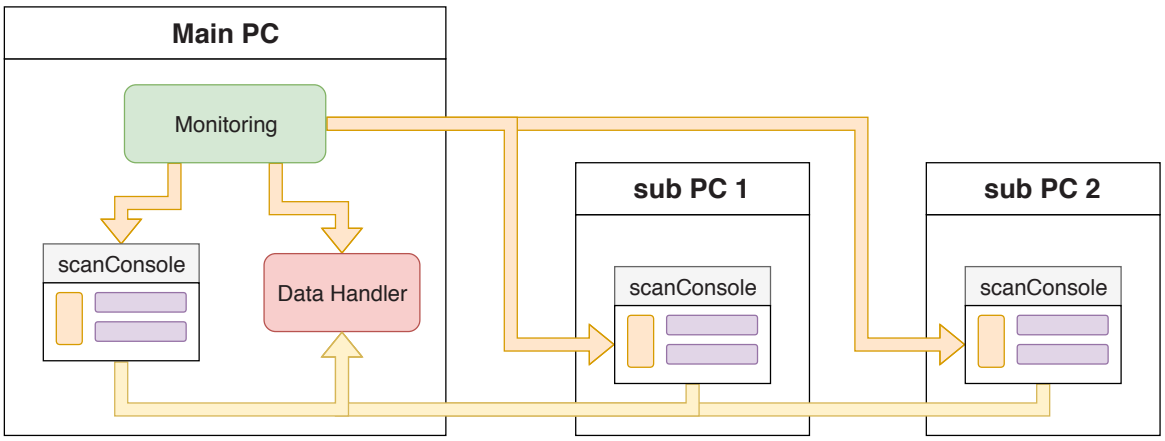


Figure 5.1: Concept for scanning multiple HW-controllers on multiple PCs.

can be seen in Figure 5.1. In addition to this concept, the YARR-software still lacks a way of handling exceptions within HW-controller threads. Currently, errors within the SPEC driver for example are handled by throwing an error message and then terminating the entire process. This is obviously not an ideal state, as an error in one SPEC driver would result in the abortion of all other HW-controller threads. This means, a way of communication between the main method of the thread and sub-methods needs to be established. Currently, the most reasonable way of achieving such communication is via return values of all subordinate functions. This, however, requires much effort and patience, as the YARR-software is very complex and branched widely. Thus, such changes could not be made in the context of this bachelor thesis. There are also some ways to expand the logging class introduced in this thesis. Primarily, it should be implemented into the entire software, as it is currently only implemented into SPEC- and emulator-related files as a proof-of-concept. After this, a useful addition is the ability to set different verbosity levels to the output from different classes. During development on the software, it is often necessary to get debug output only from specific parts of the software. Currently, all output would have to be given at a DEBUG verbosity level. Needless to say, this makes for a very confusing and unclear output.
It might be possible to test the use of commercial logging classes for use in the YARR-software as well. Most of these logging classes are thread safe and allow for output to multiple destinations, including logging files. However, there are currently no known loggers that have the option to turn off output from specific classes. Thus, further elaboration on the logging class introduced in this thesis appears to be the most reasonable step for future development on the YARR-software.

Bibliography

[1] L. Evans. *The CERN Large Hadron Collider: Accellerator and experiments*. 14 August 2008. http://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08001/pdf.

[2] A. L. Read. *The discovery and measurements of a Higgs boson*. 6 December 2013. http://iopscience.iop.org/article/10.1088/0031-8949/2013/T158/014009/pdf.

[3] The ATLAS Collaboration. *ATLAS Forward Detectors for Measurement of Elastic Scattering and Luminosity*. 17 January 2008. https://cds.cern.ch/record/1095847/files/lhcc-2008-004.pdf.

[4] The ATLAS Collaboration. *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*. 15 June 2018. https://cds.cern.ch/record/2285585/files/ATLAS-TDR-030.pdf.

[5] T. Heim. *Performance of the Insertable B-Layer for the ATLAS Pixel Detector during Quality Assurance and a Novel Pixel Detector Readout Concept based on PCIe*. 28 August 2015. http://inspirehep.net/record/1503616/files/CERN-THESIS-2016-085.pdf.

[6] T. Wittig. *Slim Edge Studies, Design and Quality Control of Planar ATLAS IBL Pixel Sensors*. April 2013. https://cds.cern.ch/record/2305498/files/Dissertation.pdf.

[7] The ATLAS Pixel collaboration. *ATLAS Pixel detector electronics and sensors*. 24 July 2008. http://iopscience.iop.org/article/10.1088/1748-0221/3/07/P07007/pdf.

[8] The ATLAS IBL collaboration. *Production and integration of the ATLAS Insertable B-Layer*. 16 May 2018. http://iopscience.iop.org/article/10.1088/1748-0221/13/05/T05008/pdf.

[9] J. Wolf. *C++ - Das umfassende Handbuch*, pages 719–772. Galileo Computing, 2014.

[10] *XPressK7 Specifications*. 29 August 2018. https://www.reflexces.com/products-solutions/other-cots-boards/xilinx/xpressk7.

[11] *KC705 Manual*. 10 July 2018. https://www.xilinx.com/support/documentation/boards_and_kits/kc705/ug810_KC705_Eval_Bd.pdf.