A multithreaded data acquisition dll

by
Simon Sebastian Humpohl

Bachelor's Thesis in Physics

presented to

The Faculty of Mathematics, Computer Science and Natural Sciences
at RWTH Aachen University

Department of Physics, Institute II C

July 2014

supervised by

Hendrik Bluhm

Abstract

This thesis is a manual and documentation for a C/C++ library with the purpose to speed up gathering and processing the data of the state of a two-electron spin qubit. Due to the very fast experimental manipulation time of this qubit in the nanosecond-timescale and the need to operate it in a feedback loop the library heavily makes use of multithreading to acquire and process the data on-the-fly. In this way the host application can control the experiment with minimized dead time. The set of operations includes downsampling and calculating the repetitive average signal as well as higher level operations e.g. histogramming the downsampled values which are used in single-shot measurements.

Contents

Part I

Introduction

Chapter 1

Motivation and Overview

Today, multi-core processors are widely spread, not only in super computers which rely on multiprocessing for a long time but also in common home computers, laptops, smart phones and other consumer products[1]. In order to use this additional potential for computationally expensive problems applications must be designed in a different way compared with traditional single threaded programs.

The problem to solve was to acquire the measured data of a electron spin qubit using a PCIe Data Acquisition Card and process it while allowing the host application the preparation of the next measurement in parallel. Therefore the library which solves that problem uses the benefits of multithreading not only for performance, but also to allow reducing the experiment's dead time due to parallel data procession.

Qubits are the quantum computer analogon to the classical bit. Quantum computing offers more efficient solutions for a set of problems like famous Shor's algorithm which allows integer factorization in polynomial time[1] or a native simulation of another quantum mechanical many-body system like Richard Feynman supposed already in 1982 [2]. For quantum computation the control over single qubits and the implementation of quantum gate operations are crucial.

The original task was to improve an existing C library which used the Windows-API threading functions. The old library had issues with memory leakage and bad performance in some situations so I rewrote the whole source in C++ to make use of STL containers which help to avoid memory leakage and the C++11 multithreading features for operation system portability and object oriented access.

To understand the design of the library it is useful to know what the intended use of it is so I will firstly describe the experiment and the typical work flow, followed by the operations that are dictated by this in the rest of Part I. The library itself is explained in Part II beginning with important properties of the PCIe card in section 3.1. The rest of the chapter 3 describes the library detached from implementation details followed by

[1] `http://store.steampowered.com/hwsurvey/cpus/?sort=pct`

chapter 4 where these details are given and where important data types, functions and the API are documented.

Chapter 2

Background

2.1 The experiment

The library gets its data from an AlazarTech PCIe card which measures the read-out voltage of a double quantum dot(DQD) used as a election spin singlet-triplet qubit. Such a qubit uses the singlet $|S\rangle = \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle)$ and the triplet $|T_0\rangle = \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle)$ state of the coupled electrons in the DQD as computational states. Two-electron spin quantum qubits are promising candidates for quantum computing since universal control over the qubit has been achieved in times far shorter than the rate of decoherence[3]. The PCIe card has replaced the read-out with an oscilloscope which was much slower since between each measurement the recorded raw data was transfered to the computer and processed there which costed a lot of time.

To determine if the DQD is in a triplet or singlet state the potential is detuned as seen in figure 2.2. Electrons in singlet state will now both be localized in one dot(2,0) while the pauli principle forces triplet state electrons to remain in the (1,1) occupation. Due to pertubation f.i. from phonons there will be transitions between the states during measurement time and due to energy relaxation the triplet decays in the lower energetic singlet state in the relaxation time.

A sensing quantum dot coupled to one of the dots as seen in figure 2.1 measures the charge in the proximal dot and is included in a tank circuit so its resistance is proportional to the damping of the LC-resonator, when excited. Analysing the reflection of an applied rf-signal at resonator frequency as depicted in figure 2.3 allows determining this damping which leads to the charge and the associated qubit state.

2.1.1 Measurement organization

A measurement cyle as seen in figure 2.2 is controlled by a control pulse applied on the gates which i.a. detunes the potential. These pulses are grouped in pulse groups that are repeated several times in one scanline. A exemplary scanline configuration repeats a pulse

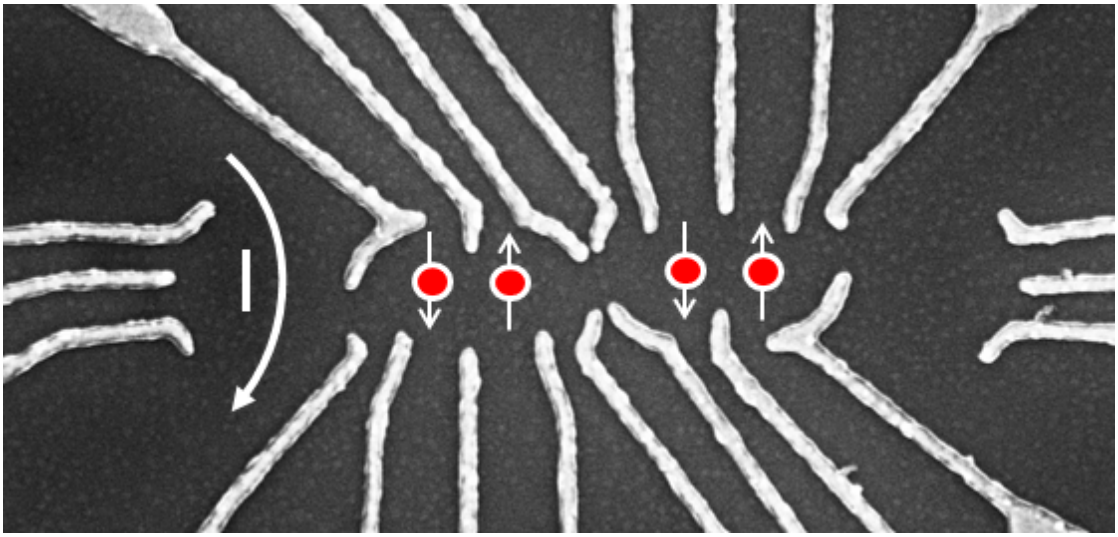


Figure 2.1: A SEM picture of two GaAs double quantum dots with sensing dots similar to the used one.

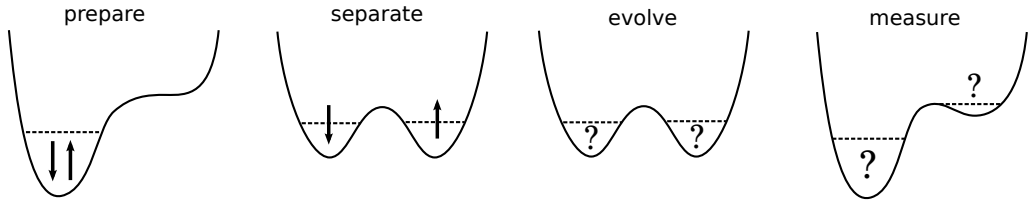


Figure 2.2: The general process cycle consists of singlet(2,0) preparation, electron separation, an evolution phase and the measurement. This cycle has a typical length of about 4 µs is repeated many times.

group that contains 100 pulses for 1536 times. In each pulse group the evolution time spent in the (1,1) configuration is ramped from 0 ns to 100 ns to observe the time dependency of the qubit state i.e. each evolution time has 1536 corresponding measurements. I will refer to this configuration in the next section when the operations are explained to clarify what they do.

One measurement process consists of multiple equally configured scanlines like the one above that each are started by an external trigger. Since the card records the whole scanline the actual signals from the read-out phases have to be identified in the recording. This is done applying user defined masks on the data. These masks contain the start, the end and the period of the measured signal in the raw data which is identical to the start and end of measurement in one measurement cycle and its length. Typical values are e.g. for start 2 µs, for the end 4 µs and for the period 4 µs which have to be converted to the corresponding sample count with the usual sample rate of 100 MS s^{-1}. Figure 2.4 shows how such a mask is applied on raw data.

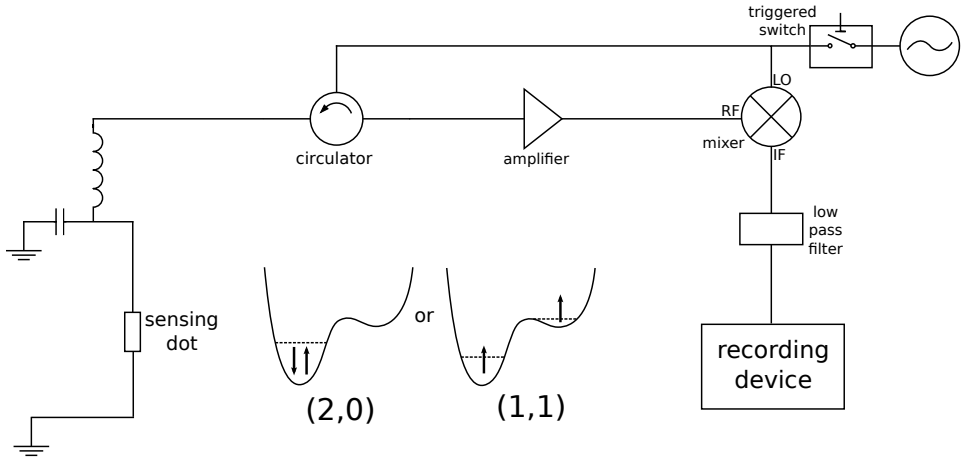


Figure 2.3: A simplified model of the experiment the library is intended to use with. The occupation of the DQD influences the damping of the tank circuit.

2.2 The library operations

The library offers an extendable set of operations that consist of operations working with the raw, unprocessed data and operations which utilize the results of the raw data operations. Currently, the only operations which are applied on the raw, unprocessed data are downsamling and repetitive signal averaging (see below). These are therefore computed before the others. At the moment, the other operations are only applied on the results of the downsampling operation but it is possible to use the repetitive signal averaging results as well if desired. Each operation has at least one mask it refers to but can in general work with an arbitrary number of masks. This may in future be a necessary feature for correlation between different masks or channels.

Downsampling The downsampling operation reduces each signal to one value, its average. In this way the noise is removed and the information about the qubits state can be accessed. To do this the average value can be compared the voltages typical for a singlet or triplet states. With the exemplary configuration from the last section one gets $2\,\mu s \cdot 100\,MS\,s^{-1} = 200$ samples per signal. These 200 samples are downsampled to one value. So if the scanline contains 1536 pulse groups à 100 pulses the downsampling operation produces $100 \cdot 1536 = 153600$ values, which is 400 times less then the raw data.

Repetitive signal averaging In this operation the average signal over all pulse repetitions of a whole scanline is computed to get the average dynamic of the quibit during read-out time. Repetitive signal averaging allows measuring the relaxation time of the qubit in

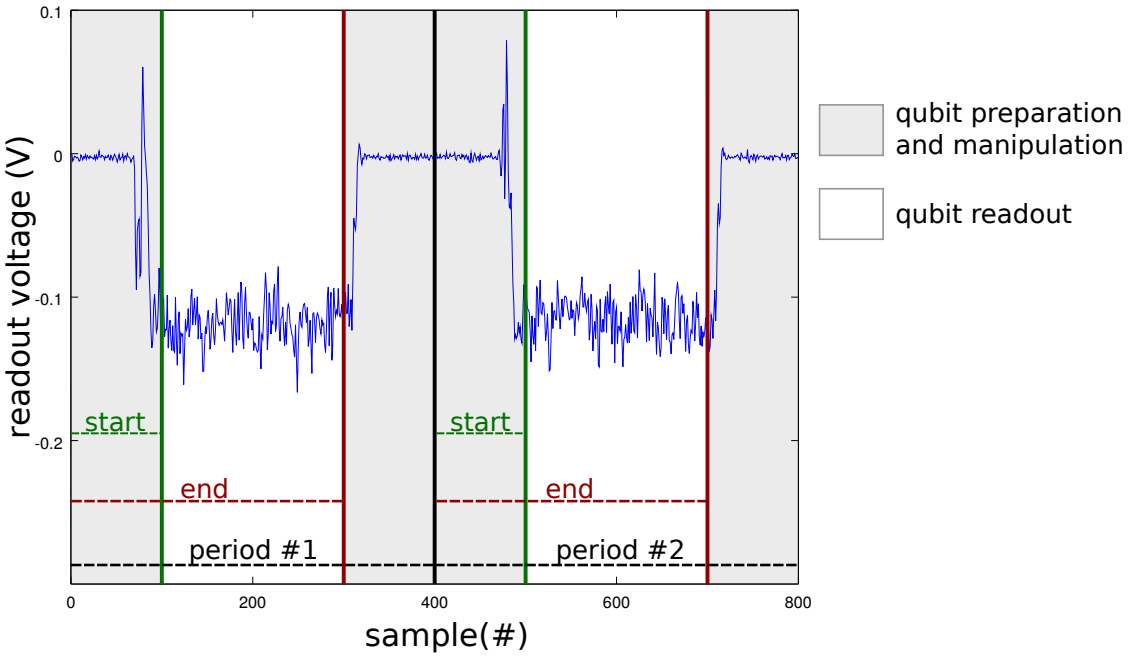
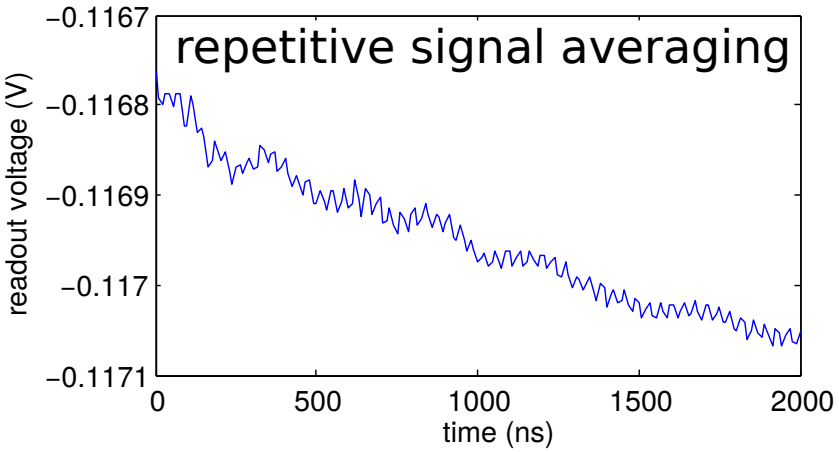


Figure 2.4: Raw data of thwo measurement cycles with the mask identifying the read out signals. The start and end and given relatively to the begin of each period.

read-out process by directly fitting an exponential decay curve to the average signal. How such a signal can look like can be seen in figure 2.5.

Figure 2.5: The results of the repetitive signal averaging operation of a typical scanline configuration showing how the dot decays statistically from triplet to singlet over a measurement.



Raw moment The k-th raw moment of a sample is the mean value of the k-th power of the values X_i in contrast to the central moment which refers to the deviation from the mean. It can be used to estimate the raw moment of the samples distribution.

$$\mu_k = \frac{1}{n}\sum_{i=1}^{n} {X_i}^k$$

It is a common experimental setting to use N different pulses and repeating them M times in one scanline. The raw moment operation takes N as an argument and computes the desired raw moments using every N-th downsampled value as sample i.e. one gets the k-th raw moment for each pulse.

If one wants to compute the first moment with the configuration from section 2.1.1 the raw moment operation averages over the 1536 repetitions of the pulse group so you can get the average read-out value μ_1 for each of the 100 pulses like in figure 2.6. To get its variance Δ one needs to compute the second raw moment μ_2 and use $\Delta = \mu_2 - {\mu_1}^2$. The variance can be used to calculate autocorrelations that are i.a. needed for analysis of the nuclear spin bath[4] but this operation is not implemented yet.

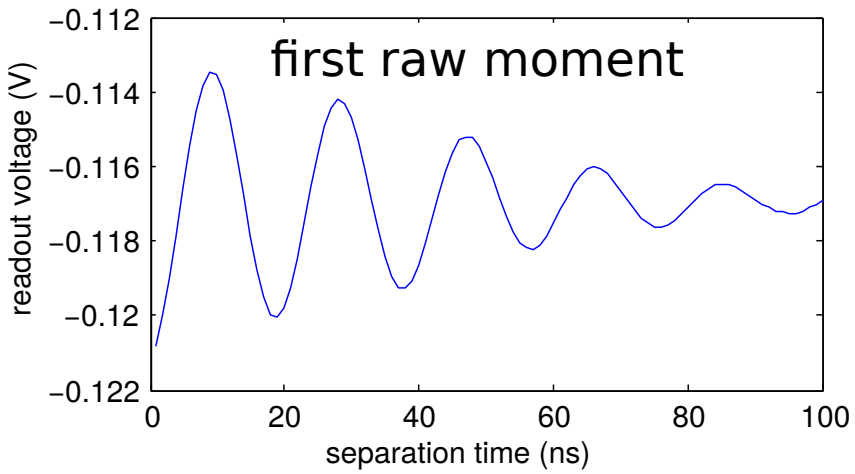


Figure 2.6: The first raw moment with N = 100. The separation time for evolving was ramped in steps of 1 ns over 100 signals for 1536 times so each point can be identified with a separation time. You see how the system oscillates between the two states over time.

Histogram To get the typical voltages for triplet and singlet states you can fill the downsampled values in a histogram. The triplet and singlet states will each form a peak, as seen in 2.7, so you can for example normalize the voltage or find a threshold voltage parting the voltages in triplet and singlet voltage [5].

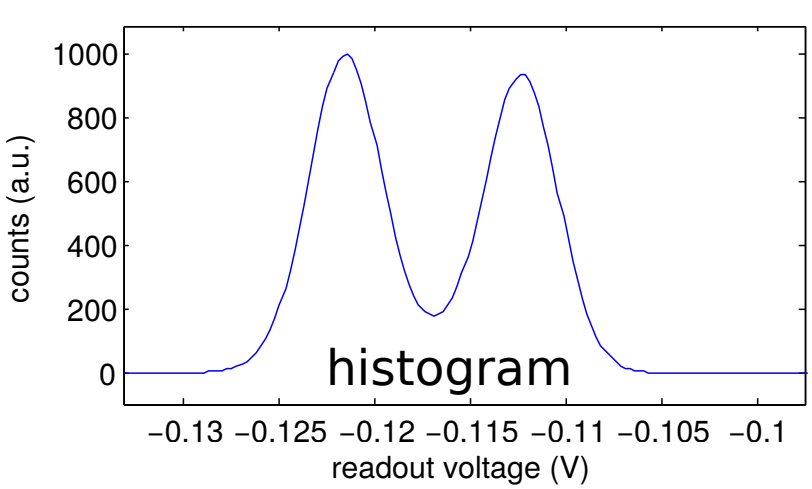


Figure 2.7: The result of histogramming the downsampled values. The right peak can be identified with the triplet and the left with the singlet state.

Part II

The library

Chapter 3

In words

Now I will describe the library itself and how it works to give an orientation for the source chapter as well as a short description of the used hardware.

3.1 The hardware

The library was developed with a ATS9440 AlazarTech - 14 bit PCI Express Digitizer which AlazarTech suggests for OCT, radar, ultrasonics, spectrometry and RF signal recording[6]. This library uses the asynchronous DMA (Direct Memory Access) triggered streaming mode of the card so the card waits for an external trigger and then records data until it is aborted. DMA means that the card copies the data to main memory directly without involving the CPU.

The board provides the data in coherent parts of user defined length. To get these, the library has to supply the card with memory blocks where the card can transfer its data to. This process is called posting buffers. The buffers are handled following the first in first out (FIFO) principle so the library needs to track which buffers where posted in which order. For further understanding the lecture of section "Steaming data across the bus" of the Alazar board manual is very helpful. The AlazarAPI itself is explained in the SDK Programmer's Guide[7].

3.2 How the library works

The user configures and starts the acquisition with a single function call of AtsConfigure-Measurement(...). Besides the technical configuration, e.g. the sample rate and the buffer size, the function takes masks and operations as arguments. The library controls which masks and raw data operations are needed and marks them like illustrated in figure 3.1. As mentioned above operations do always have at least one raw data operation applied

on a mask they refer to, for example the histogram operation needs the downsampled values.

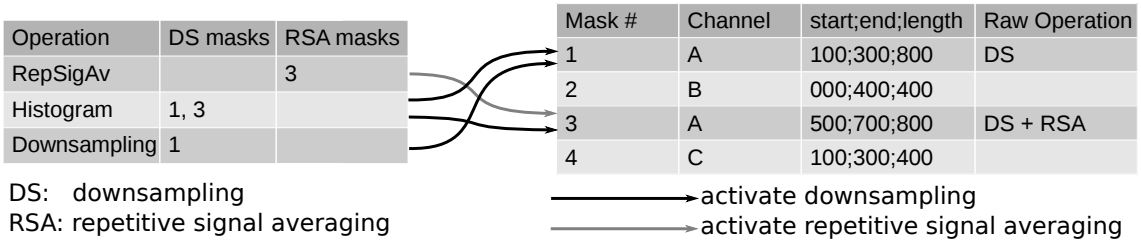


Figure 3.1: Example illustrating how the operations Repetitive Signal Averaging, Histogram and Downsampling activate the raw data operations the rely on. Repetitive Signal Averaging and Downsampling will simply forward the results whereas the Histogram operation will fill the downsampled values from Mask 1 and 3 in a histogram.

Then the acquisition thread seen in figure 3.2 is started, begins supplying the card with buffers and arms the card by telling it to wait for a trigger signal with an AlazarAPI call. When the card is triggered it will start recording and transferring the recorded data to the provided buffers. As soon as the card signals the acquisition thread that the next buffer is ready the acquisition thread assigns this buffer to a computation thread which does the downsampling and repetitive signal averaging operations. Since a scanline in general contains multiple buffers the library has to merge the buffer-wise computed results of the operations to one scanline result as soon as the whole scanline data is collected. The higher level operations that need these results, like histogramming, are executed afterwards by the merging thread. Since the acquisition thread only collects buffers and assigns the tasks it spends most of the time waiting for buffers which ensures that there will be no buffer overflow due to the lack of posted buffers.

The scanlines are stored in a queue ordered by their acquisition time. As soon as the merging thread has finished it marks the current scanline as ready so it can be accessed by the interface functions. For result access, call AtsGetOneResult(...) to get the results from one operation, AtsGetAllResults(...) to get the results of all operations or AtsGetRawData(...) to get the raw data. These functions will return the results of the first scanline in the queue once it is ready. When all results of the scanline were fetched AtsPopScanline(`HANDLE boardHandle`) has to be called to remove this scanline from the front of the queue so the next scanline can be accessed with the fetching functions. If it is not called they will always fetch the same results.

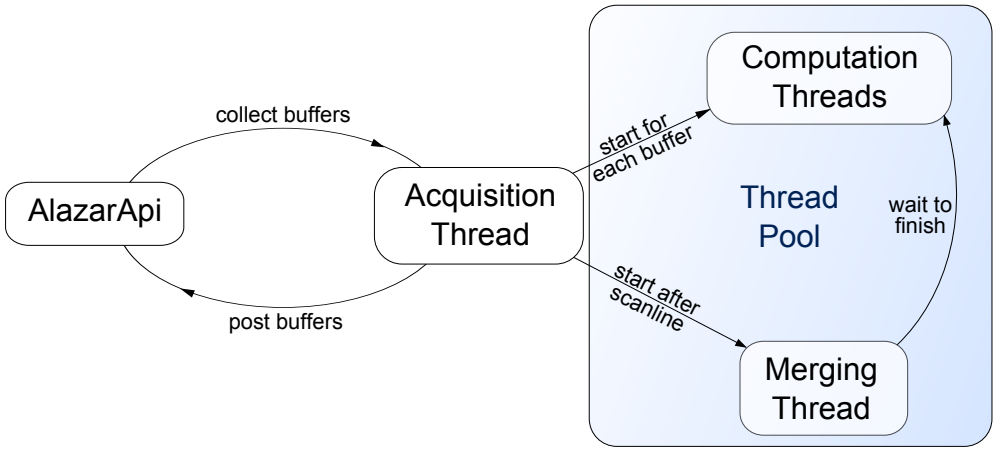


Figure 3.2: A model of the threads and their relations. The computing and merging threads live in the thread pool while the acquisition thread itself is separated.

3.2.1 Features and important implementation details

The central object is an instance of the BoardManager class. It manages the configuration of the card, the data acquisition and holds the final results. It is stored as a global managed pointer in "interface_functions.cpp" so it can be accessed through the C-style interface functions. The working threads are started at manager construction and stored in a thread pool. In this way a thread does not to be started when it is needed which can be considered to be a costly operation.

The buffer pool To avoid memory allocation at acquisition time some buffers are pre-allocated and filled in a buffer pool realized as a queue. As soon as the raw data is no longer needed the buffers are fed back to the pool so the function that posts buffers can reuse them. If there are not enough buffers in the pool there will be new ones allocated. The buffer pool does also have a configurable maximum size so the surplus buffers are deleted. There is also an option to deactivate the buffer pool at runtime.

The scanline pool The scanline pool is similar to the buffer pool with the difference that there is no real need for it yet. It was implemented with a triggered continuous streaming mode in mind so the whole acquisition can be asynchronous from the host application. The scanline pool cannot be deactivated.

The clean mode When the clean mode is activated, which is default, the card is reset and the buffers are reposted after each scanline. This is done to delete the data that

eventually is still an the on-board memory and that would be at the beginning of the first buffer of the next scanline.

The logger I wrote a simple thread safe logging class for easy debugging and a rudimentary channel functionality. The channels are created via pre-processor macros and can either be logged to the log file or be dumped. Dumped channels are completely optimized out at compile time since the pre-processor defines an empty log function for dumped channels. The active log channels are written to "logpath/averagingdll_N.log" where "logpath" can be set via AtsSetLogPath(`const char* path`) and N denotes the rotation count. Each log contains a mutex lock to protect file access so excessive does may block other threads decreasing the performance of the library.

3.3 General programming remarks

The library is written with a MATLAB Windowsx64 host in mind but can in principle be used by any application. One main goal during the development process was to make the library extendable and portable so it also could be can be used on a Linux platform since there are no Windows-API calls in the library itself. The only use of them was made for the implementation of the status window.

I am in favor of long describing variable and function names. Even if it takes longer to code it makes understanding of the code for future maintainers a lot easier. I did not make use of `using namespace std;` because I find it more convenient if STL-containers are clearly recognizable.

3.3.1 The status window application

I did write a status window application which uses the shared memory features of Windows to access status variables of the library. It shows the number of posted buffers, buffers in the pool, scanlines in use and scanlines in the pool as well as it shows all masks with their activated operation(s). Since the code of it is not well tested and I do not understand the memory sharing comlpetly this is not further documented in this thesis. The data sharing has to be turned on at compile time by defining the pre-processor constant `WITH_STATUS_WINDOW` in order to use the status window.

3.3.2 Wrapping mex-function

Since the operations may have a arbitrary number of controlling parameters they can not be passed directly as C structs from MATLAB to the library. Therefore I wrote a so called mex function, which is a c++ function that uses MATLAB headers and can read MATLAB structs, to convert the structs that define the operations in MATLAB to the corresponding classes in the library. The implementation itself has a few workarounds because I am not familiar with MATLAB's mex library so it is definitely not final and not futher documented in this thesis.

3.3.3 Source code organization

I divided the code in source and header files in different folders and there is a source and a header file for each class like it is standard. The `class` BoardManager source is split in a public and a private file and the interface functions are divided in a general file for the functions that do not need a `BoardManager` instance and the instance specific files. The Alazar and TTMath headers are in subdirectories of the include folder.

3.3.4 Extending the library

The main requirement for extendability was to make it easy to extend the set of operations. To add an operation that works with the results of downsampling or repetitive signal averaging you have to modify "operations.h", "operations.cpp" and for use in MATLAB the mex-file. The other operations can be used as examples when performing these steps:

1. Read the section about `class` PostProcess.
2. Declare the operation as a class that inherits from `PostProcessing` in "operations.h".
3. Declare a function that takes the arguments for creating an instance of this class and returns a pointer to it. This function has to be exported using the predefined `ATS_DLL_API` macro in "operations.h".
4. Implement the class in "operations.cpp".
5. Implement the function in "operations.cpp".
6. Extend the mex-file so it can identify your MATLAB struct for the operation and call the creation function with the correct arguments.

Chapter 4

In code

Now I will come to the actual source code starting with the integer data types and the auxiliary classes for masks and operations. These are used by the board manager which is the central class in this library. The chapter is closed by the description of the interface functions which can be used from MATLAB. Figure 4.1 shows the overall program flow and helps to sort in the following classes and functions.

4.1 Data types and class descriptions

4.1.1 Simple data types

The library uses the typedefs `SN` for signed and `UN` for unsigned integers. The N denotes the number of bits in the integer and can be any power of 2 from 8 up to 128. The `U128` is realized via the TTMath library[1] and the smaller ones are build in types. By having the integer bit-length at hand it is easier to avoid integer overflows. For performance reasons the usage of large integers as `U128` is and should stay as rare as possible since they do not fit in a x64 register.

The card records data in 14bit resolution and stores them in an `U16` with two zeros at the end. The library output values are always `U32` integers except when the raw data is asked. They are converted in the functions that do downsampling and repetitive signal averaging.

4.1.2 `struct` Mask

The `Mask` contains the mask for the raw data and the information if the downsampling and repetitive signal averaging operations shall be performed with their results. The actual mask contains the starting point, the end point and the period of the masks signal in the data stored in units of samples. Since the results are computed buffer-wise they have to

[1] TTMath is a small template library which provides big numerical data types: `http://www.ttmath.org/`

be merged as soon as all buffers are ready. This is done by calling the `mergeResults()` method.

```
struct MATLABMask
{
        U32 begin;
        U32 end;
        U32 period;
        U32 hwChannel;
};
struct Mask
{
        U32 begin;
        U32 end;
        U32 length;
        U32 period;

        U32 hwChannel;

        // final results
        std::vector<U32> downsampledData;
        std::vector<U32> repetitiveAverage;

        // computation results for each buffer
        std::vector< std::vector<U32> > downsampledDataPerBuffer;
        std::vector< std::vector<U64> > repetitiveAveragePerBuffer;

        // merge the results. A false return indicates something has gone wrong
        bool mergeResults(  const size_t& buffersInScanline,
                            const U32& samplesPerBufferPerChannel);

        Mask(const MATLABMask& m,
            const U32 buffersPerScanline,
            const U32 downsamplesPerBuffer,
            bool computeRepetitiveAverage );
};
```

The constructor uses its arguments to perform all necessary memory allocations so there will be none later on. The sizes of `downsampledData` and `repetitiveAverage` indicate if the associated operation will be performed. If booth are empty the Mask is ignored while computing buffers. The `mergeResults()` merges the buffer-wise results i.e. it averages over the average signals of each buffer while storing the intermediate results in a `U64` vector to avoid an integer overflow. The downsampled values are attached to each other in the order of the buffers.

4.1.3 class PostProcess

The class PostProcess is an abstract class providing the interface for all operations. If you want to extend the library by an additional operation you must implement it a class that inherits PostProcess as described in section 3.3.4. It is mostly stored as an PostProcessingOperation which is a typedef for a std::unique_ptr<PostProcess>. This class is exported since the mex-function handles pointers to it.

```
class PostProcess
{
public:
    //Use the data to compute the operations results.
    //Return false if an error occurs.
    virtual bool processData(const std::vector<Mask>&) = 0;

    //Set the booleans of the needed operations at their masks to true.
    virtual void activateNeededOperations(std::vector<bool>& downsample,
                                          std::vector<bool>& average) = 0;

    //Return the result of the operation. Throw exception if not ready.
    virtual const std::vector<U32>& result() = 0;

    //Return a pointer to a copy of this operation.
    virtual PostProcess* copy() const = 0;
};
```

The activateNeededOperation(...) function sets the the booleans of the according mask index in the needed operations vector to true. With this method only those masks are applied and downsampled that are needed later on. The processData(...) method performs the operation on the according masks. It gets passed all masks so the interface is independent of the actual mask needed. The result is implemented as a function because the downsample and repetitive average operation forward to the according mask and do not store the results for them selves since their results are already calculated in Mask::mergeResults(). The actual computation of the results of the other operations happens in processData(...).

4.1.4 class Buffer16

The Buffer16 class is a simple typedef std::unique_ptr<U16>. In this way there is no memory leakage of buffers and they are easier to identify in the code.

4.1.5 `struct` Scanline

The Scanline class combines the raw data buffers, the masks and the post processing operation in one object as seen in figure 4.2.

```
struct Scanline
{
    std::shared_future<bool> ready;

    std::vector< std::pair<Buffer16, std::future<bool> > > bufferStorage;

    std::vector< Mask > masks;

    std::vector< PostProcessOperation > operations;

    Scanline(const std::vector<Mask>& Nmasks,
             const std::vector<PostProcessOperation>& Noperations,
             size_t buffersPerScanline);

    //this is done to avoid copying if not explicitly wished
    Scanline copy() const;
    Scanline(Scanline&& s);

    Scanline(const Scanline&) = delete;
};
```

`ready` is the return value of the function/thread completing the scanline. If an error occurred during computation it is set to false otherwise it is true. The buffers are each stored together with a `std::future<bool>` which shows if the buffer computation has completed without errors in `bufferStorage`. `Scanline` allocates all needed memory except for the buffers at construction. The memory used by the Scanline is meant to be reused, so the copy constructor is deleted to avoid unintended copying instead of moving. For copying you have to use the copy method which calls the constructor with the right arguments and moves the so generated object. In this way the buffers will not be copied since this would illogical.

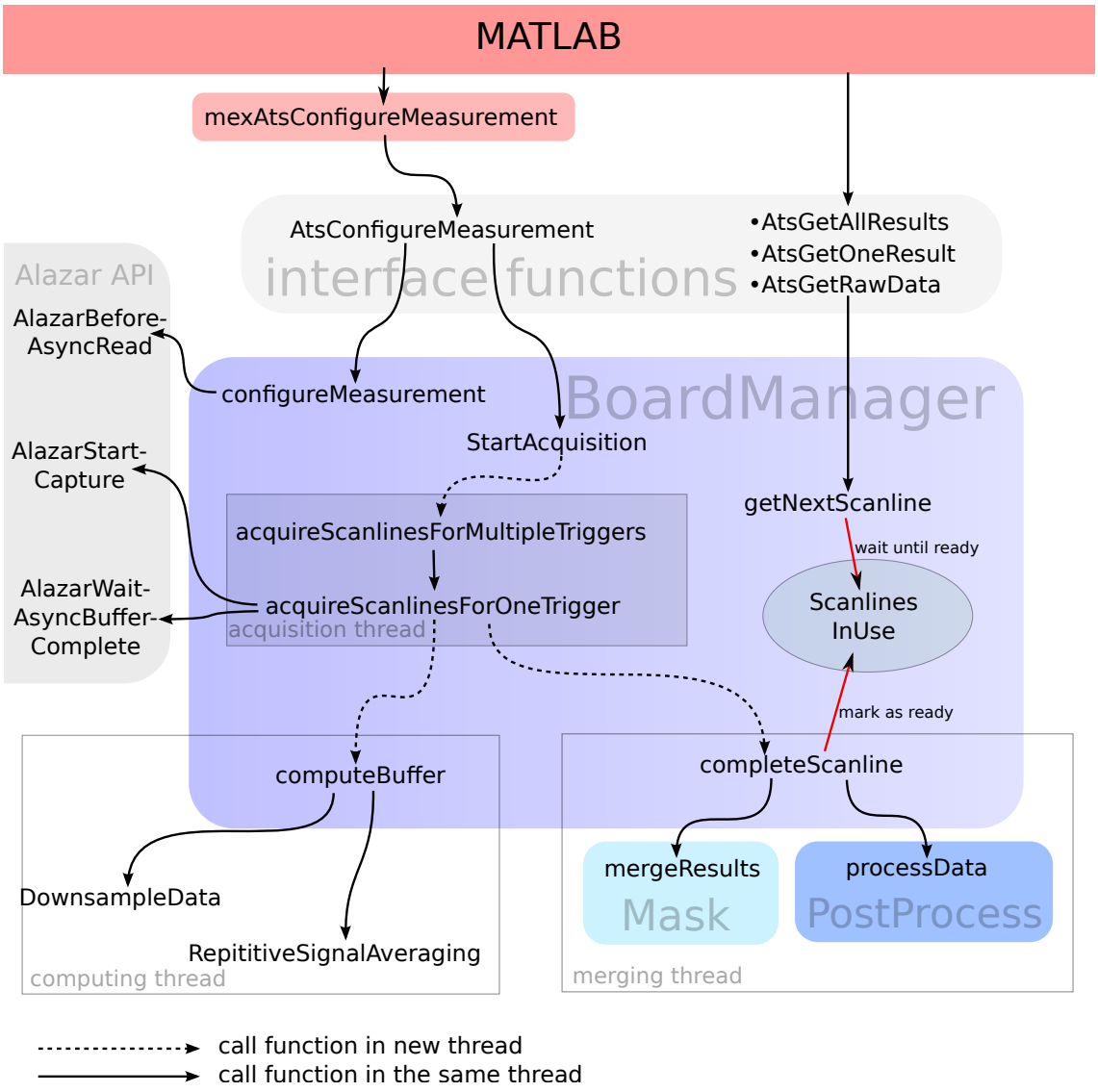


Figure 4.1: Program flow for configuration and acquisition without clean mode. The interface functions wrap the BoardManager who configures the PCIe card, acquires and processes the data and provides the results. The computing and merging threads live in the board managers thread pool and the acquisition thread is independent. The mex-function convertes MATLAB structs to C++ classes.

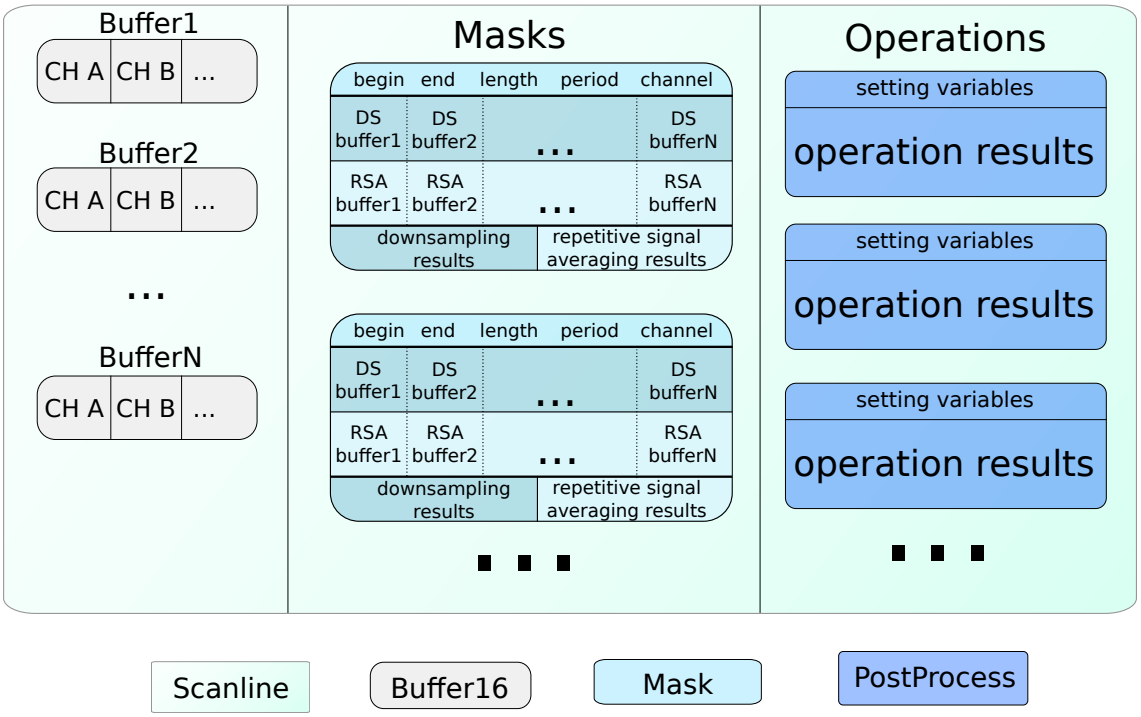


Figure 4.2: This figure shows what data is stored in the Scanline class and how it is organized. The data in the buffers is used to compute the buffer-wise results of the downsamling and repetitive signal averaging operations which are merged in one result by the merging thread later on. The operations may refer to these results by forwarding to it as the repetitive average operation does or store the results for them selves.

4.2 The BoardManager

`BoardManager` is the central class in this library. It holds the configuration variables and scanlines, configures the card and performs all computations. In this section I will describe how the BoardManager class implements the important functionalities by listing the involved member variables and methods and explaining their relation. There is no explicit description of every method as for the interface functions in 4.3 because these methods do work closely together. Some settings are static members so they can be set instance independent and are persistent until the library is unloaded.

4.2.1 Misc functionalities

The clean mode The clean mode implementation is fairly simple. After each scanline `cleanMode` is checked and if it is true, `cleanCardPerpare()` gets called. This function aborts the card, clears the posted buffers and posts new ones.

```
static bool cleanMode;
private:
void cleanCardPerpare();
```

Listing 1: Variables and functions for the clean mode.

The thread pool The private member variable `TaskQueue` holds all computing threads and tasks. It is realized as a regular member of the BoardManager so it is automatically and cleanly deleted if the BoardManager is deleted. The size is determined by `std::thread::hardware_concurrency()` which should give a reasonable number of threads the machine can compute in parallel.

For implementation I used a class by Jakob Progsch[2] published under zlib/libpng license and added a method to restart all threads.

Finding the channels in the buffers The purpose of the member variables `std::vector<U16> relativeChannels` and `std::vector<U16> absoluteChannels` is to connect the position of a channel in the buffer with the actual channel index. They are computed in `configureMeasurement(...)`. The easiest way to understand their function is an example like table 4.1.

[2] `https://github.com/progschj/ThreadPool` commit 20061c5c7fc64f9bfe2f5852ad5551e4adf379ad

Recorded Channels	A	D		
`relativeChannels`	0	x	x	1
`absoluteChannels`	0	3		

Table 4.1: The content of `relativeChannels` and `absoluteChannels` when the channels A and D are recorded. The x show that the channels B and C are not recorded and their numerical value is `std::numeric_limits<U16>::max()`.

4.2.2 Buffer handling

```
static bool useBufferPool = true;
static size_t maximalBufferPoolSize = 110;
private:
std::mutex bufferPoolMtx;
std::queue<Buffer16> bufferPool;

void moveBufferToPool(Buffer16& buffer)
void moveBufferToPool_noMutexLock(Buffer16& buffer)
```

Listing 2: Member variables and methods implementing the buffer pool.

The buffer pool As their names indicate useBufferPool determines if the buffer pool functionality is used and maximalBufferPoolSize sets the maximal buffers in the pool. If a buffer gets added to the pool via the `moveBufferToPool(...)` method while the maximum pool size is reached the buffer will be deleted. `bufferPoolMutex` should always be locked when `bufferPool` is accessed to avoid data races. The `moveBufferToPool_noMutexLock(...)` method leaves this responsibility to the user and is only implemented to allow filling in multiple buffers with one mutex lock for performance reasons.

```
static size_t bufferPostIntervall = 2;
static size_t maximalNumberOfPostedBuffers = 100;
static size_t minimalNumberOfPostedBuffers = 10;

private:
size_t aimedPostedBuffersSize;
std::queue<Buffer16> postedBuffers;

void postBuffers(size_t bufferNumber = 1);
void fillupPostedBuffers( size_t desiredSize = 0 );
```

Listing 3: These member variables and functions control the posting of buffers.

Posting buffers The values of `minimalNumberOfPostedBuffers` and `maximalNumberOfPostedBuffers` are only used to calculate the aimedPostedBuffersSize in the measurement configuration. The acquisition thread will try to have aimedPostedBuffersSize buffers posted to the card during an acquisition.

The posted buffers are organized in a queue because the card fills them first in first out and expects the buffers are collected in the order they are posted. There is no mutex here because the ONLY thread that may access postedBuffers is the acquisition thread. This means that the postBuffers and fillupPostedBuffers functions may only be called by this thread.

4.2.3 Scanline handling

```
static size_t preallocatedScanlines = 2;

const Scanline& getNextScanline(U32 maximalWaitinMilliseconds);
RETURN_CODE popScanline();

private:
std::unique_ptr<Scanline> referenceScanline;

std::mutex scanlineMovementMtx;
std::list<Scanline> scanlineBuffer;
std::list<Scanline> scanlinesInUse;
```

Listing 4: Scanline maintaining member variables and methods.

The reference scanline is constructed in `configureMeasurement(...)` and every other scanline is a copy of it which ensures that all scanlines are equal. The actually used scanlines are stored in `std::list` containers because so they can be easily moved between `scanlineBuffer` and `scanlinesInUse` very fast and without copying. At acquisition start the buffer is filled with `preallocatedScanlines` scanlines whose default value 2 is sufficient for the current work flow. At the beginning of each scanline a `Scanline` is moved from the buffer to `scanlinesInUse` and gets filled with data and computation results. The first scanline in `scanlinesInUse` can be accessed via `getNextScanline(...)` which throws an exception if there is no `Scanline` to return or if an error has occurred during scanline acquisition. If the scanline is no longer needed `popScanline()` has to be called which moves the scanline back to the buffer.

Always lock `scanlineMovementMtx` when moving scanlines!

4.2.4 Measurement configuration

```
public:
void configureMeasurement(
                U32 NsampleRate,
                U32 NbuffersPerScanline,
                U32 NsamplesPerBufferPerChannel,
                std::vector<Mask>&& NoperationMask,
                std::vector<PostProcessOperation>&& Noperations,
                U16 NstoreRawData);
```

This function sets the BoardManagers configuration variables so it has to be called before StartAqcuisition. The masks and the operations have to be passed via the move semantic so they can be move-constructed.

The provided `Mask` objects have to be activated before, since the method will only compute the channel mask for the board by checking which masks are activated and merging them into the rawDataMask. Beside this the absoluteChannel and relativeChannel vectors are created. These are needed to find a channels position in the buffers.

```
private:
std::vector<Mask> masks;
std::vector<PostProcessOperation> operations;

U32 samplesPerBufferPerChannel;
U32 buffersPerScanline;
U32 sampleRate;
U16 channelMask;

U32 bufferByteSize;
U32 bufferSize;
```

Listing 5: These variables hold the configuration of the current measurement.

4.2.5 Acquisition thread control

`StartAcquisition(...)` allocates memory for buffers and computation results, prepares the card and starts the acquisition thread by calling `acquireScanlinesForMultiple-Triggers(...)`. When the method is called and the acquisition thread is the method aborts it. To prepare the card `cleanCardPrepare` is called. The acquisition thread itself is started via `std::async` command with the return value of the acquisition itself being stored in `acquisitionReturnValue` so it can be controlled if there was an error.

```
public:
void StartAcquisition(U32 triggerCount, U32 scanlinesPerTrigger);
bool AcquisitionThreadRunning();
void AbortAcquistionThread();
RETURN_CODE AbortAcquisition();
```

Listing 6: These public member functions provide control over the acquisition thread.

`acquisitionReturnValue` is also used by `AcquisitionThreadRunning()` to check if an acquisition currently is in progress. The user can abort the acquisition with `AbortAcquisition()`

```
private:
std::shared_future<RETURN_CODE> acquisitionReturnValue

std::atomic<bool> abortAcquiring

std::mutex acquisitionMtx
```

Listing 7: Member variables that are used internally for acquisition thread control.

which will stop the acquisition thread, call `AlazarAbortAsyncRead()` and free the buffers posted to the card. The internal control of the acquiring thread checks `abortAcquisition` after each buffer so aborting will probably last a buffer length. `acquisitionReturnValue` holds the acquisition threads return value as soon as it has finished.

4.2.6 The data processing functions

```
RETURN_CODE acquireScanlinesForMultipleTriggers(
                                            size_t triggerCount,
                                            size_t scanlinesPerTrigger);
RETURN_CODE acquireScanlinesForOneTrigger(std::deque<Scanline*>
                                            scanlinesToFill);
bool computeBuffer(Scanline& scanline, const U32 bufferId);
bool completeScanline(Scanline& scanline, U16 maximal_wait_milliseconds);
```

Listing 8: The functions that are used internally to acquire and process the data.

The acquisition thread starts in `acquireScanlinesForMultipleTriggers(...)` and loops in there over the number of triggers. For each trigger `scanlinesPerTrigger` scanlines are moved to `scanlinesInUse` and a `std::deque` filled with pointers to each of them is cre-

ated. This is the argument for `acquireScanlinesForOneTrigger` which will iterate over this queue and fill each scanline in it with data.

I admit that this way of handling it is not nice and I advice future editors of this code to work with lists and take the `Scanline`s from the scanline buffer at the beginning of each iteration in `acquireScanlinesForOneTrigger`. The mutex call necessary for this should not be to expensive.

Immediately after a buffer returns from the card `computeBuffer(...)` is enqueued with this buffer in the task queue. As soon as all buffers of the scanline are collected `completeScanline(...)` is enqueued to. `computeBuffer(...)` iterates over all masks in the `Scanline`, checks which operations have to be done and does them while `completeScanline(...)` waits till all buffer wise computation have finished and then calls the `Mask::mergeResults()` method of each mask and the `PostProcess::processData(...)` method of each operation.

The `getNextScanline()` checks the return value of `completeScanline(...)` to determine if the computations have finished yet and if they were successful.

4.3 C-style interface functions

In this section the interface functions of the library are documented. The dll-API has two types of functions that are declared in the AtsAverage header file "AtsAverage.h". The general functions found in "general_interface_functions.cpp" modify options that are independent of a particular measurement and are stored as static members of `BoardManger` or the logger. The other functions rely on a `BoardManager` instance which is a global variable stored together with these functions in "interface_functions.cpp". They control the measurement, handle the computation results, abort the running acquisitions and/or reset the card their embedding in the program flow is described in section 3.2.

The `BoardManager` is stored in a `std::unique_ptr<BoardManager>` to allow easy resetting and cleanup. With a bit of work the library should be able to manage more than one card. My suggestion is using a `std::map` with `HANDLE`s and unique pointers for manager storage.

All functions return a `RETURN_CODE` defined in "AlazarError.h" which shows if anything went wrong. More detailed information may be found in the log file.

4.3.1 General settings

AtsSetLogPath(`const char* path`) sets the path the logger will write the next log to. The log is written to "path/averaglingdll_N.log" with N being the log rotation count.

AtsRotateLog() switches to the next log file. The rotation is done automatically when `AtsConfigureMeasurement(...)` is called.

AtsUseBufferPool(`bool mode`) sets the flag for the usage of the buffer pool. If set to false buffers will be allocated and freed for every Scanline.

AtsSetCleanMode(`bool mode`) sets the flag for the clean mode. If set to true the card will be reset after each scanline.

AtsSetBufferPostIntervall(`size_t bufferPostIntervall`) The buffers are posted in groups of `bufferPostIntervall` buffers. The hope behind this is that maybe the transfer to the buffers from card is faster if the post buffer function gets called more seldom. But there was no benchmarking done.

AtsSetTriggerTimeout(`U32 timeout_in_milliseconds`) Sets the time the acquisition thread waits for the trigger before aborting.

AtsSetBufferLimits(`size_t lowerBound, size_t upperBound`) sets the bounds for the number of buffers posted to the card. The exact number is determined by the buffer size.

AtsSetPreallocatedScanlines(`size_t preallocatedScanlines`) Sets the number of scanlines which are allocated before the measurement. The only need to change this, was for a continuous streaming mode which is not implemented yet.

4.3.2 Measurement configuration and results

AtsConfigureMeasurement(...)

```
//function arguments:
HANDLE boardHandle,
U32 samplesPerBufferPerChannel,
U32 buffersPerScanline,
U32 sampleRate,
U32 scanlinesPerTrigger,
U32 triggerCount,
const MATLABMask* Nmasks,
size_t maskCount,
PostProcess const * const * Noperations,
size_t operationCount
U16 rawDataMask
```

The function configures the card and allocates most of the needed memory before starting the acquisition. The call from MATLAB has to be wrapped by a mex function to convert MATLAB structs for operations and masks to C++ classes. This function configures the BoardManager and starts the data acquisition via its member functions `configureMeasurement(...)` and `StartAcquisition(...)`. The `MATLABMasks` get converted in `Mask` objects and the operations in `Noperations` are copied.

The first approach of passing the masks and operations as `std::vectors` did not work because the binary memory layout of STL objects may not be compatible between different compilers/compiler settings and therefore these should be used in a library interface.

AtsPopScanline(`HANDLE boardHandle`) has to be called if the information of a scanline is no longer needed. If it is not called the `AtsGet*()` functions will always return the same data.

AtsGetAllResults(...)

```
//function arguments:
HANDLE boardHandle,
U32* apiBuffer,
U32 bufferLength
```

This function writes the results of all operations in `apiBuffer` in the order the operations were passed. The raw data is will not be copied by this operation. If the summed length of the computation results does not match `bufferLength` the function returns `ApiFailed`.

AtsGetOneResult(...)

```
//function arguments:
HANDLE boardHandle,
U32 operationIndex,
U32* apiBuffer,
U32 bufferLength
```

This function writes the result of the operation with index `operationIndex` in `apiBuffer`. If the results length does not match `bufferLength` the function returns `ApiFailed`.

AtsGetRawData(...)

```
//function arguments:
HANDLE boardHandle,
U32* apiBuffer,
U32 bufferLength,
U16 channel
```

This function writes the raw data of the specified channel in `apiBuffer` if the channel was set in the `rawDataMask` argument of AtsConfigureMeasurement(...). The data is not converted to 32-bit integer but copied directly from the `U16` buffer. This means that `bufferLength` must be the number of samples divided by two since every `U32` containes two values.

4.3.3 Aborting functions

AtsAbortAll(`HANDLE boardHandle`) calls `AtsClearAll()` and `AtsInitialize(...)` so the library can be used afterwards.

AtsAbortAcquisition(`HANDLE boardHandle`) aborts the current acquisition by calling the abortion function of the manager. Scanlines stay in memory.

AtsClearAll() deletes the `BoardManager`. This means that all threads are stopped and all memory is freed.

Part III

Conclusion

Conclusion

4.4 Comparison between old and new version

The previous version of the library was implemented in C and most things where done directly in the API functions with the whole code being in one source file. It provided only the downsampling and repetitive signal averaging operations and the total number of buffers was configured at compile time. The thread structure was similar to the one used in the new library but was controlled with the WIN32 multi-threading API.

Now, the API functions only wrap the board manager functions which makes it easier to change the implementation or the API independently. The switch to C++ was motivated by the wish to use STL containers and the possibility to implement multi-threading through language features instead of external libraries. I divided the source in multiple source files so the code structure follows the structure of the library.

Besides these implementation details the new library now offers a extended AND extendable set of new operations. A transitional version of the library which is capable of downsampling, repetitive signal averaging and computing the first raw moment is already in use at the experiment.

An important aspect concerning stability is that the library waits for each buffer for most of the time. This means that buffer overflows that occur because the on-board memory of the card is full are not caused by the library but by slow transfer from on-board memory to main memory.

4.5 Outlook

The only thing left to do is implementing additional operations as they are needed in the experiment. For operations that rely on downsampled values this is quite simple as suggested above. The only problematic thing could be new operations that rely on the raw data. In such a case the system how raw data operations work may have to be refactored.

4.6 Last words

The rewrite of the library was successful. But only time will tell if extending the library by a third party works. Good luck.

Bibliography

[1] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124 –134, nov 1994. Cited on page 3.

[2] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982. Cited on page 3.

[3] Sandra Foletti, Hendrik Bluhm, Diana Mahalu, Vladimir Umansky, and Amir Yacoby. Universal quantum control of two-electron spin quantum bits using dynamic nuclear polarization. *Nature Physics*, 5(12):903, October 2009. Cited on page 5.

[4] T. Fink and H. Bluhm. Noise spectroscopy using correlations of single-shot qubit readout. *Physical Review Letters*, 110:010403, Jan 2013. Cited on page 9.

[5] C. Barthel, D. J. Reilly, C. M. Marcus, M. P. Hanson, and A. C. Gossard. Rapid single-shot measurement of a singlet-triplet qubit. *Physical Review Letters*, 103:160503, Oct 2009. Cited on page 9.

[6] Alazar Technologies Inc. Ats9440 user manual. Feb 2011. Cited on page 13.

[7] Alazar Technologies Inc. Alazartech SDK Programmer's Guide - version 6.1.0. 2013. Cited on page 13.

Appendix

1 Required C/C++11 knowledge

The source code containes a few features in C/C++ that are not a common part of a beginners tutorial. So I will give a few short explanations of these features and hints for further reading.

The multithreading of the library is based on the threading functionality of C++ introduced in the C++11 standard. The usage of C++11 brings the advantage of code portablility and defined behavior for multithreading as well as a few new usefull features and STL classes.

1.1 C-style bit(shift) operations

As the board configuration function `AlazarBeforeAsyncRead` takes a binary channel mask as one of its arguments I will give a short explanation how to modify single bits in an integer. Bitshift operators shift the bits of an integer in memory to the left (<<) or to the right (>>) which is the same as multiplying or dividing by two. So `1 << 5` shifts the bits of the one 5 positions to the left so the result is `1 << 5` $= 100000_2 = 32_{10} = 2^5$. The `&` and `|` operators perform an bitwise AND or OR comparison between the operands: `00111 & 01101 = 00101` respectively `00111 | 01101 = 01111`.

```
int a, n;

a = a | (1<<n);  //set the n-th bit of an integer a
a |= (1<<n);     //same operation in short

bool is_bit_set = a & (1<<n);        //check if the n-th bit is set
```

Listing 9: Example: set and read single bits.

1.2 `std::unique_ptr` and `std::move`

A `unique_ptr` contains only a pointer of the associated type. It calls the destructor of the object it is pointing to on destruction or when it is assigned an other pointer. This eliminates memory leaks caused by a missing delete/free statement. To transfer the ownership of a pointer you have to use the move semantic. If there were two `unique_ptr`'s owning the same object both of them would call its destructor leading to undefined behavior and a segfault[3].

1.3 Lambda expressions, `std::thread` and `std::mutex`

In this code lambda function syntax only is used to start threads and all threads are started with lambda functions[4]. Understanding this is only neccessary if one wants to edit the code in these regions. The use of `std::mutex` is more common in the source since mutexes are responsible for avoiding race conditions that occure when two ore more threads access the same variable.

In the cases I used lambda expressions I wrote all used variables in the captures field `[...]` instead of simply capturing all implicitly by reference `[&]`. Variables that are captured by reference may out of scope if the function is executed causing undifined behavior or segmentation faults.

[3] `http://www.cplusplus.com/reference/memory/unique_ptr/`

[4] Since it is a quite complicated field I will not explain it here. As usual cppreference.com has a good article `http://en.cppreference.com/w/cpp/language/lambda`.

```
int a = 0;
int b = 3;
std::thread t(
[a,&b]{
    /*do stuff*/
    a = 3;        //changes the local copy of a
    b = a + 5;    //changes the actual value of b
    callFunction(a,b);
});

/*do other stuff in parallel */
a = 4;
b = a + 3; //dangerous since t also modifies b -> undefined behaviour

if(t.joinable())  //checks if t holds a thread
    t.join();     //waits for t to finish
```

Listing 10: Example: what happens without `std::mutex`.

As you see in this code snippet both threads t and the main thread modify b so the possible outcomes of this are 8, 6 and 11. To avoid this you can use the `std::mutex` and optionaly `std::lock_guard<std::mutex>`. A lock guard locks a mutex on construction and unlocks it on destruction. I always used lock guards in my code for two reasons:

1. The risk of forgetting an unlock and causing a deadlock of the program is far smaller.

2. The mutex gets unlocked if an exception is thrown since the lock guard lives on the stack and will go out of scope then.

```
std::mutex mtx;
int a = 0;
int b = 3;
std::thread t(
[a,&b]{
    std::lock_guard<std::mutex> lock(mtx); // lock mutex
    b = a + 5;

    /* the mutex would be unlocked if an exception is thrown here
    ** since the lock_guard is destroyed then */
    callFunction(a,b);

    //here the mutex is unlocked as the lock object is destroyed
});
//save because no exception possible:
mtx.lock(); //lock mutex
b = a + 3;
mtx.unlock(); //unlock mutex

if(t.joinable())
    t.join();
//b is definitely 11 now
```

Listing 11: Example: `std::mutex`.

1.4 `std::future` and `std::async`

`std::future` and `std::shared_future` store the result of a parallel running computation. They have two important methods: `valid()` returns if the object is associated with a thread and `get()` waits for the thread to finish and returns the return value. A `std::future` can only be read out once whereas `std::shared_future` can be accessed multiple times.

`std::async` creates a thread that executes the argument and returns a `std::future<return_value>`.

```
std::future<int> result;
result.valid(); //returns false
result = std::async( // this will be done in parallel
[]{
    /*do stuff*/
    if(error)
        return -1;
    return 0;
});
result.valid(); //returns true
return result.get(); //waits till the asynced thread
                     //finishes and returns its return value
```

Listing 12: Example: `std::future` and `std::async`.

List of Figures

Acknowledgements

I would like to thank my supervisor and head of the group Hendrik Bluhm for giving me the opportunity to work on this subject and give a practical contribution to scientific research. Although my decision to rewrite the code was risky he gave me the freedom to try. While working in the Bluhm group I experienced a motivated atmosphere and I want to thank everyone there for the fun I had. In special I thank my advisor Tim Botzem for his support, explanations and for proofreading about 2^{64} drafts of this thesis. I want to thank Pascal Cerfontaine for some very helpful conversations that broadened my view. I also thank Patrik Bethke for providing me with hints about software and websites that turned out to be very useful.

I am very thankful to my parents and family for supporting me and enabling be to follow my interests. I also thank my friends and fellow musicians for the friendship and for providing distraction when needed.

Thanks to the bands Bolt Thrower, Atheist and Deep Purple for the sound track that accompanied me while writing the library and this thesis.

Statement of authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All sources of information have been specifcally acknowledged in all conscience.

Aachen, July 7, 2014 Simon Sebastian Humpohl