Bachelor Thesis

Development and Implementation of
a Central Trigger System for TrbNet-based systems

Manuel Penschuck

Institut für Kernphysik
Johann Wolfgang Goethe-Universität

First examiner:	Prof. Dr. Joachim Stroth
Second examiner:	Dr. Ingo Fröhlich
Advisor:	Dr. Jan Michel

December 10, 2012

Selbständigkeitserklärung

„Hiermit erkläre ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderen fremden Texten entnommen wurden, sind von mir als solche kenntlich gemacht worden. Ferner erkläre ich, dass die Arbeit nicht – auch nicht auszugsweise – für eine andere Prüfung verwendet wurde.“[1]

Manuel Penschuck

Hattersheim, den December 10, 2012

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Historical Motivation and State-Of-The-Art

Since its early days, experimental nuclear physics has been strongly influenced by the technological progress and has often had to cope with limitations of the equipment obtainable. This obviously applies to the detectors involved: Until the availability of reliable photo multipliers (1920s), most well-known experiments had to rely on photosensitive films that used chemical processes producing directly visible structures as an indirect confirmation of the presence of particles and radiation.

The Rutherford experiment (1909[1]), for instance, had to capture patterns that emerged from the scattering of at least several million of incoming silver atoms due to the limited sensitivity of the films in combination with the cross section between silver and gold atoms. Later, it became possible to observe single reactions, e.g. using bubble chambers (1952). Such a chamber visualises the trajectory of a charged particle interacting with a superheated liquid inside a magnetic field. In order to quantise the observations, photos were taken and manually analysed. The conditioning of the fluid as well as the (semi-) manual read out strongly bound the feasible event rates and thus only allow the investigation of common phenomena which do not suffice any more to understand details of the standard model.

With new developments of the detectors, the limiting factor of the feasible data rate began shifting towards the data acquisition. MWPCs[2] (1968) and TPCs (1974) can be interpreted as a more versatile replacements of bubble chambers. Instead of optically detecting local phase transitions of some fluid, one directly measures the electrical potential difference between adjacent wires induced by charged particles traversing the volume. Its main advantage is the fully automated electrical read out which initially allowed for event rates of ≈ 1 KHz (about three orders of magnitude higher than any other particle detector with comparable capabilities at that time) [19].

The MWPC is an early example of a triggered detector type. To achieve a sufficient spacial resolution, a large number of channels is required. In a free running operation mode, additionally a high sampling rate is needed in order to capture short events. The resulting raw data rate exceeded the capabilities of cost-efficient data storage and processing systems. As a solution, a trigger was employed to select potentially interesting samples: Early MWPCs used analogue comparators to detect channels above a given threshold. Such a discriminator produces a digital signal, which instructs the data acquisition system to read out the sensor.

---

[1] First year of operation

[2] **M**ulti-**W**ire **P**roportional **C**hamber, evolved from the non-proportional spark chambers (1930s)

Level	Input [events/s]	Suppression factor	Technology
1st	$4 \cdot 10^7$	400	ASICs and FPGAs
2nd	$1 \cdot 10^5$	30	$\approx$ 1K CPUs/GPUs
3rd	$3 \cdot 10^3$	15	$\approx$ 3K CPUs/GPUs

Table 1: Rough performance estimates of ATLAS trigger system. Values may vary depending on the experimental settings. Note that layer 3 requires 3 times more machines, despite a 30 times lower event rate. Hence the computational efforts per decision are 100 time higher.
After stage 3, an average event produces 1.6 MB data, leading to a data rate of 320 MBs^{-1}. If stage 1 worked with full granularity, a data rate of 640 TBs^{-1} needed to be processed. As this rate cannot be processed, only a selection of channels with increased granularity is monitored by the trigger layer 1 yielding an input rate of only 3 $TBits^{-1}$ [4] [3].

If a detector does not rely on external information to identify possible events, it is referred to as a self-triggered system. Alternatively, a central device may trigger several detectors in one experiment. This central trigger is typically connected to a subset of detectors that allows for an easy event identification. Many large setups are centrally triggered because of the inherent synchronisation of independent subsystems. Hybrids between both schemes also exist [8].

The continuously increasing performance of new data acquisition systems allows for better event identification algorithms, more indirect and ambiguous sensors as well as higher resolutions – both temporal and spacial. The process of triggering itself remains – of course – a compromise between rejection efficiency and the complexity required to achieve it. Nowadays, pipelined multi-level triggers further improve the decision accuracy. Each level propagates only a (favourably) small subset of its inputs to the next stage and thus allows to increase the computational cost per decision from layer to layer. In this scheme, lower levels are designed to rather produce a large amount of false-positive, as this wrong decision can be corrected by a higher instance while false-negatives are lost.

Even if there is no absolute model, most multi-level triggers are implemented in a fashion similar to the ATLAS experiment (2009), which was chosen as an example because of its enormous data rates (see table 1):[3]

- Level 0 is usually incorporated into the detectors itself and deals with the raw data. It primarily suppresses background noise. Depending on the implementation it might also perform an analogue to digital conversion.
- Level 1 uses this information to detect potential events. Depending on the sensor type, the afore-mentioned primitive analogue discriminators can be assigned to both levels.

[3]In chapters 2 and 3 aspects of other experiments, especially the HADES experiment, are discussed in greater detail

In recent setups, level 1 is based on digital inputs and implemented via special purpose processors, such as ASICs or FPGAs. They mostly perform massive parallel but computationally cheap operations, such as simple pattern recognition with at most integer arithmetic. The level 1 trigger typically uses only a subset of data available, e.g. a reduced spatial granularity and an only small number of detectors.

- Level 2 often manages the event building, i.e. it gathers related values from different detectors. If the previous decisions were based only on a fraction of data, this complete information may lead to another, more educated trigger decision.

- Level 3 highly depends on the physics investigated and usually involves the reconstruction of particle trajectories using a mathematical model fitting. While lower stages concentrate on direct information, such as the amplitude, position and coincidence pulses, the fitting results in a higher accuracy of derived values, such as the energy, trajectory and type of particles.

1.2 About this Work

The aim of this work is the development and implementation of a central trigger system for small experiments. It primary targets the TRB3, a general purpose trigger and read out platform containing five FPGAs with fast interconnects and a large number of IO ports linked to up to four add-on cards. The board can operate in a stand-alone mode, in a group of several TRB3s or in a heterogeneous system, e.g. in case of an upgrade of an existing experiment.

The board is designed to communicate via the TrbNet protocol (see section 2.2) which was developed for the read out upgrade (2010) of the HADES experiment (operational since 2002) at the *GSI Helmholtzzentrum für Schwerionen Forschung* in Darmstadt, Germany [11].

The existing TrbNet Central Triggering System is highly optimised for the complex HADES detector which requires an expensive dedicated trigger board for performance reasons (see section 2.3). However, for many smaller experiments a more cost-efficient TRB3 based approach suffices. The new system implements a full hardware description of all trigger functionality[4] and offers flexible synthesis and run time configurations.

The device is controlled and monitored via a remote software solution designed to interoperate with the existing TrbNet programs. The tool chain further offers a comfortable and easy to use graphical user interface, implemented via advanced web technologies.

[4] As discussed in chapter 3, the functions are comparable with those described for the trigger layer 1 in the previous chapter

The client application can be executed on any computer with a recent web browser connected to a web server executing the CTS tool chain.

After discussing a number of mandatory prerequisites, this work analyses the hardware (chapter 3) and software design (chapter 4), as well as the interaction of both components. By design, the document is neither intended as a reference nor a documentation (which will be available under [9] at the end of 2012). It focuses on the overall picture and considers structural challenges and the chosen approaches as well as alternatives.

2 Prerequisites

2.1 Programmable Logic

In digital electronics there are many ways to implement a required functionality. Almost all methods produce a complex behaviour by connecting a number of simpler building blocks.

Most commonly, a (micro-)processor is used to execute a program, i.e. a series of atomic instructions. Due to its sequential nature, the processor's hardware structure is fixed and the functionality arises at runtime, defined only by the instructions and their order in the program flow.

Thus, in a very simple model, the run-time of a program depends on the complexity of the problem and increases linearly with the amount of instructions needed. While this is acceptable for many applications, an ordinary processor can hardly cope with highly parallel tasks under strict timing constraints.

As will be discussed in the following chapters, the trigger component of the CTS is an instance of such a case: It possesses a number of inputs that need to be monitored synchronously with little computational efforts, but with a high sampling frequency. Even if modern processors may be able to handle that task, the exact execution time becomes unpredictable due to the complexity of the chip. It therefore is necessary – or at least easier and more accurate – to design a dedicated circuit that meets the requirements. Instead of a PCB[5] with hard-wired logic blocks, programmable logic – more precisely an FPGA[6] – is used.

An FPGA is an electronic component containing a matrix of so-called *logic blocks* that are connected via a complex on-chip network. Slightly simplified, a logic block possesses a number of input bits B (typically 4 to 6) and a LUT[7] that defines a boolean output value for every input vector. Thus, one block is capable of computing any B-ary combinatorial function. An optional *D flip-flop* at the block's output can be used to synchronise its value to a clock signal. In order to implement common components, such as shift registers or adders, more efficiently, many FPGA families offer special types of blocks – often grouped together into bigger logic units.

The function of the logic blocks and the routing between them can be configured. The resulting circuit is limited only by the number of logic blocks available and the propagation time through the network.

---

[5] **P**rinted **C**ircuit **B**oard

[6] **F**ield- **P**rogrammable **G**ate **A**rray

[7] **L**ook-**U**p **T**able, i.e. the definition of a function $f_B : \{0,1\}^B \to \{0,1\}$

Board used on	LatticeECP2/M-100 [14] TRBv2 with CTS-Addon	LatticeECP3/150EA[15] TRBv3
No. of LUTs	95K	149K
Inputs per block (B)	4	4
Ram Blocks (18 kbit each)	288	372
No. of IO pins	416	586 / 380

Table 2: Comparison of key features of the two FPGAs used in this work. While the ECP3 chip offers more logic blocks and memory, there is no significant structural difference. The TRB3 board uses two different footprints – a bigger one for the central chip, and four smaller packages for the peripheral units. However, internally both types are identical.

As a rule of thumb, the number of transistors (and hence the area on chip and its power consumption) required to implement a circuit on an FPGA is at least ten times higher compared to the implementation on an ASIC[8]. Furthermore, an ASIC allows for a clock speed about one order of magnitude higher than an FPGA.

The main advantages of programmable logic over ASICs are the shorter development time and the lower initial costs. To reduce the technical disadvantages, virtually all FPGAs include hard-wired features for frequently used structures, such as memories, PLLs[9], SERDESs[10], or even whole-processors.

Design tools and synthesis

Similar to the process of compiling software source code into a machine specific program, the hardware configuration is typically synthesized from a source code written in a hardware description language. Three well-known languages are *VHDL*[11], *Verilog* and *SystemC*. All of them were originally designed to model a system for simulation and verification purposes only. In all cases, the automatic synthesis became available years after the initial release of the languages and – by design – can cover only a subset of the languages' features.

Most commonly the hardware description is first translated into a generic netlist that subsequently is used to produce device specific hardware descriptions, e.g. an FPGA-configuration file or photomasks for an ASIC. This process is quite similar to the compilation of an ordinary computer program, which often involves at least one intermediate language.

While SystemC (initially released in the year 2000) is the most modern of the languages mentioned above, it is primarily used for simulation.

[8] **A**pplication **S**pecific **I**ntegrate **C**ircuit, i.e. a custom chip designed for a specific application

[9] **P**hase-**L**ocked **L**oop, a heterogenous circuit to recover and manipulate clock signals

[10] **Ser**ializer/**Des**erializer, converts a parallel data stream to a faster serial stream and vice versa

[11] **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

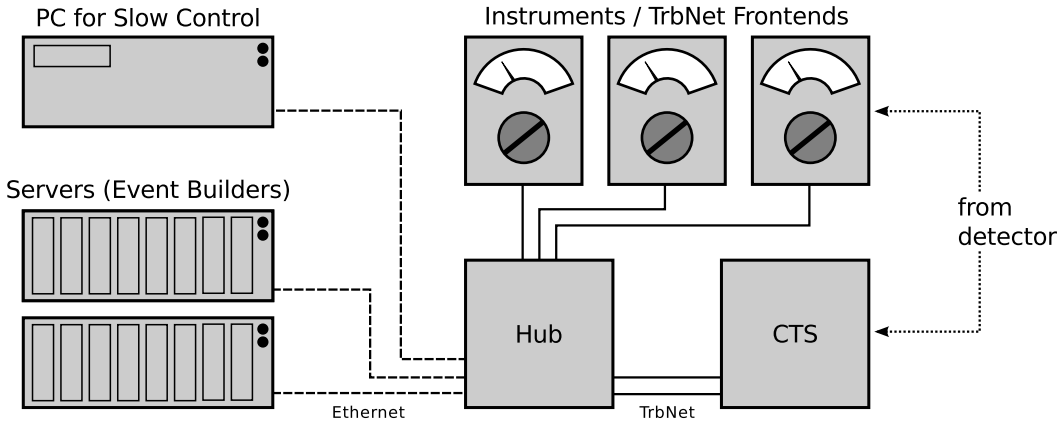

Figure 1: Example of a TrbNet based network with three data collecting frontends, connected to one hub. During the read-out process the data collected by the frontends is sent to one out of two event builders. A third computer can be used for slow control purposes. In small setups the event builder and controlling software may run on the same machine.

VHDL [10] and Verilog, however, have existed since the 1980s and share very similar features to one another. They both are specified by the IEEE and are the de facto standard languages for hardware synthesis. VHDL was chosen for this project because it has already been used for the majority of TRBv2/3 designs.

One key concept of VHDL is the so-called *entity* that can be thought of as a black box that interacts via inputs and outputs and implements a certain behaviour – just like a chip on a PCB. An entity itself can rely on other entities or independently define its behaviour. While there are a number of programming paradigms supported by VHDL, most commonly the description takes place on the register-transfer level, i.e. one uses *if-then-else*-constructs and arithmetic operations to manipulate signals and thus indirectly defines registers.

2.2 TrbNet

Many experiments – especially in modern particle physics – involve detectors that are too complex to be analysed by a single instrumentation device (see chapters 1.1 and 2.3 for examples). In those systems an infrastructure that is able to coordinate and synchronise the multitude of devices is mandatory. The TrbNet is a network protocol implemented in VHDL and designed for this application.

It offers a number of features suitable for DAQ systems, such as the trigger distribution in centrally triggered systems including an accurate time base, the readout of the data measured and slow control capabilities, i.e. the means to control and monitor parameters of the individual network devices.

The largest installation based on the protocol is the HADES experiment with slightly more than 1000 optical links. However, there are a number of smaller experiments intending to use the protocol for (parts of) their DAQ system.

Each network device has two addresses. Similarly to a MAC address used in Ethernet, there is a fixed hardware address uniquely identifying each device. It is commonly derived from uniquely tagged hardware, such as an one-wire temperature sensor on the frontend's board. In order to support systematic addresses that remain unchanged in case of a hardware replacement, all frontends additionally have a 16 bit address that can be assigned – either manually or via a DHCP-like scheme.

As depicted in figure 1, the network topology foresees only a few different device types:

- The **CTS** identifies events of physical interest –commonly from low-level detector signals–, generates the trigger information and initiates the data readout process.
- The **frontends** perform the actual measurements and gather the results.
- The **hubs** act as the network's backbone. They handle the full-duplex communication between the frontends and the CTS and optionally are able to redirect the physical relevant data to a server farm via an Ethernet uplink. While it is possible to cascade many hubs, a high fan-out count and a flat tree structure is favourable as it reduces the system's latency time and hence increases the feasible trigger rate.

The features of the network have very different demands. For instance the trigger information needs to be distributed with a minimal delay and thus should be handled via short messages. The readout, on the other hand, requires a high bandwidth and hence favours large frames that may delay the former packet type. The TrbNet approaches this dilemma by offering four independent logical channels of which only three are used.

The *LVL1 channel* has the highest priority and communicates the trigger information from the CTS to all data collecting endpoints. Once the event information packet is transmitted, the channel remains blocked until all frontends have returned a busy-release packet. As the round-trip time is well below 5 μs (which is the hard upper limit for the HADES detector), the protocol allows for a trigger frequency above 100 KHz[12], while simultaneously guaranteeing a high data integrity. Despite the low latency, the optical network suffers from a certain jitter that prevents an accurate synchronisation of the frontends. Thus, a dedicated physical line is used to propagate a time reference throughout the system.[13]

[12] The maximal trigger rate of small systems may reach 600 KHz

[13] Another approach to this problem are deterministic latency messages as implemented in the CBM DAQ. Using special transceivers and protocols, this DAQ allows for a synchronisation based on the data network with a jitter below the length of a bit in the serial data stream. Under lab conditions, a jitter of 10 ps has been demonstrated [7].

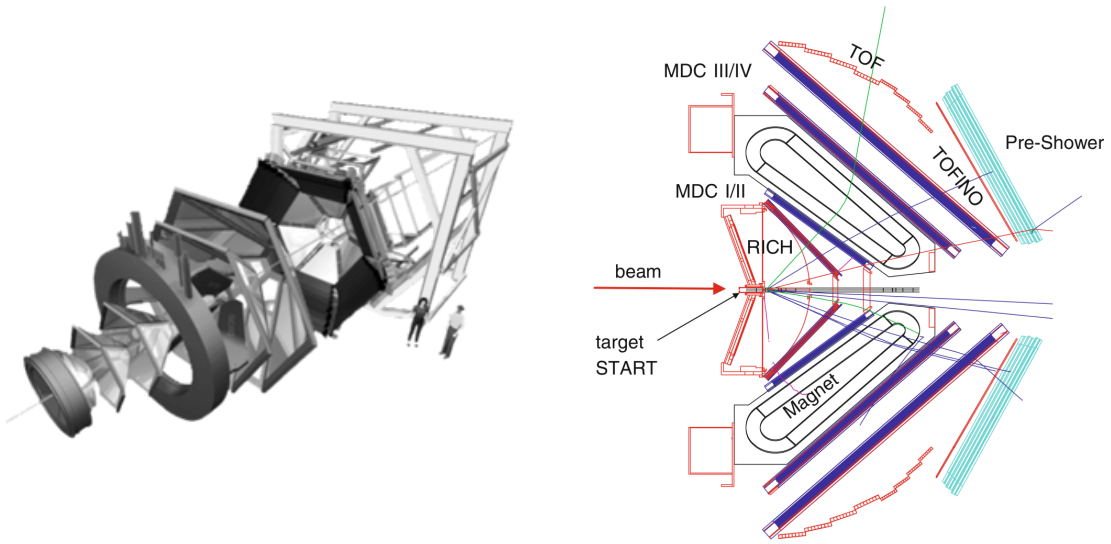

Figure 2: Overview of the HADES detector. Left: An artistic visualisation of the different subsystems [2]. Right: Components of two out of six identical segments [21]. The TOFINO has been replaced by the tRPC detector.

If an instrument receives a trigger packet, it executes its task and then writes the measured data into its frontend's local buffer. Separately issued by the CTS, this queue is later read via the *data channel*. Since the network's tree topology could cause congestions as the data travels towards the root, the hubs are able to locally reroute the data to a server farm. While up to 16 storage servers are supported, each event is processed by only one machine. As this server has to combine the data received from (possibly several) hubs, the server is called an *event builder*. The CTS, in turn, is informed only about the status of the readout process. Similarly to the busy-release scheme of the *LVL1 channel*, readout has to take place one at a time.

Each device with *slow control support* offers a (not necessarily complete) 16 bit address space with 32 bit registers. The two most common register types are *status registers*, which are read-only, and *control registers*, which allow for read/write access. All slow control requests are atomic transactions. The TrbNet allows slow control accesses on both uni- and broadcast addresses. In the later case, the responses packets from different frontends are merged while preserving the data payload and subsequently sent to the slow control server.

2.3 The HADES Detector and its TrbNet Based DAQ

The HADES[14] detector at the GSI Helmholtzzentrum für Schwerionenforschung is a general purpose spectrometer currently connected to the SIS-18 synchrotron. The heavy-ion beam with up to 2 GeV per nucleon is well suited for studying the properties of hadronic

[14] **H**igh **A**cceptance **Di**-**E**lectron **S**pectrometer

matter in a strongly interacting medium. The features of such a compressed matter are primarily observed via the di-electron (e^+e^-) decay channel of the primary particles produced in the reactions [21].

The detector's tracking system contains the two multi-layer MDC[15] systems that determine the particles' trajectories before and after the spectrometer's magnetic domain. A RICH[16] detector allows for electron identification and acts as an electron-hadron discriminator.

The outer layer of the detector contains the TOF[17] wall which measures the time of flight of a particle originating from the target and gives spatial information. The wall's lower polar angles are covered by RPCs[18] while the angles between 44° and 88° are measured via arrays of scintillator bars that are read out using a photo multiplier on each end of every bar. In order to provide a reference time, a start detector is placed in front of the target and traversed by the beam particles.

In total, the detector features ≈ 75,000 channels. In a central Au+Au collision, typically 6,000 channels are expected to fire, which sums up to 27 KB per event. At a planned event rate of 20 KHz, the DAQ, therefore, has to cope with maximum data rates of approximately 400 MBs^{-1} [18].

The HADES detector uses a two level DAQ system based on the TrbNet:

- The level 1 is completely implemented in hardware. A trigger decision is centrally made by an dedicated board and distributed to all frontends via the LVL1 busy-releasing scheme introduced in the previous section.

  The CTS board contains two FPGAs. A small but fast (800 MHz clock) chip is connected to the start and veto detectors (16 lines) and to the electronics of the TOF and RPC (one signal per segment and subsystems, i.e. 12 in total). There are eight additional inputs, so-called physics trigger, for external analogue trigger sources: While the TOF/RPC detectors offer digital signals, the physics trigger are typically linked to adder/discriminator circuits, as described in chapter 1.1. They allow for trigger criteria based on the multiplicity of coincident readings. The second FPGA handles the TrbNet stack.

[15] **M**ulti-Wire **D**rift **C**hamber, with 25,964 channels connected to the read out

[16] **R**ing **I**maging Gas **Ch**erenkov. Consists of a gas volume C_4F_{10}. Typical reaction electrons ($\beta \approx 1$) emit Cherenkov radiation, while slower hadrons ($\beta < 0.95$) do not. $28,272$ dynodes channels further provide spatial information.

[17] **T**ime **O**f **F**light

[18] **R**esistive **P**late **C**hamber; In this case a stack of three aluminium electrodes and two glass plates

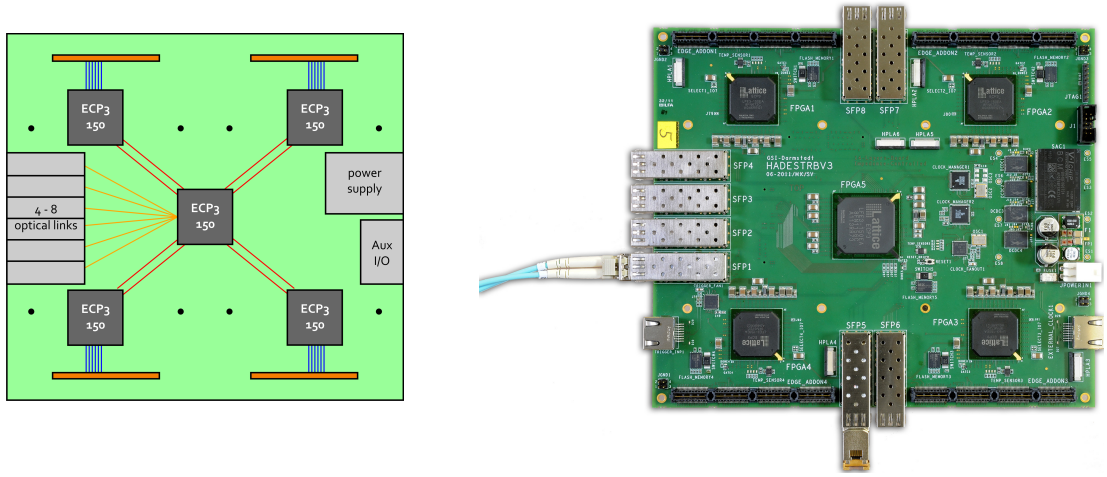

Figure 3: Block diagram and picture of the TRB3 [2].

Depending on the experiment different trigger schemes have been employed. The start detector, for instance, may solely be used to deliver a time reference or alternatively included as an independent trigger criterion in addition to the readings for the TOF wall.

In earlier experiments, a veto detector, placed behind the target, was directly used as an inhibiting trigger criterion to filter out spectators, i.e. incoming particles with no target interaction. Due to a limited performance, in more recent runs the veto detector is considered only during the offline data analysis.

- The level 2 handles the data transport. After a trigger cycle is completed, the CTS has to instruct all frontends to send their data. The responses are collected by the TrbNet hubs and sent to a server farm via twenty-five 1 GbE links connected to a 10 GbE hub.

  There are up to four standard multi-core servers each executing a number of independent software event builders that gather a complete data frame of an event and manage its permanent storage. Using a round-robin schedule, the CTS balances the load between all event builders.

2.4 TRB3

The TRB3[19] [2] is the primary target hardware of the CTS design developed in this work. While the name – given for historical reasons – can be misleading, the board is actually designed as an extensible general purpose read-out platform. Due to its compatible con-

[19] **T**DC **R**eadout **B**oard Version **3**, alternatively to emphasise the generic design: **T**rigger and **R**eadout **B**oard

nector layout on the bottom it can be used as a replacement for the TRB2 currently used in the HADES experiment. However, the TRB3 is more versatile.

In total it features five Lattice ECP3/150EA FPGAs (see chapter 2.1), organised as one central controller (FPGA 5) and four peripheral chips (FPGAs 1 to 4) each linked to an add-on connector. The on-board communication is based on the TrbNet protocol and implemented via four independent serial point-to-point links between the central hub and the outer FPGAs.

Off-board networking is supported via eight SFP sockets for TrbNet and 1000Base-X/T-Ethernet. Additionally, there are two RJ45 jacks used as trigger timebase and clock inputs. Both inputs are distributed to all FPGAs using a dedicated fan-out logic to ensure the reliable and synchronous arrival of the signal at each chip. However, most pins of the sockets are connected directly to the central FPGA and can be used for application specific features either differentially or single ended.

After power-on, the FPGAs read their configurations from flash memories that can be programmed via TrbNet. As all peripheral chips have an identical pin-out, the same configuration bit stream can be used. For that reason, the board contains five one-wire temperature sensors in order to derive an unique id for each TrbNet endpoint.

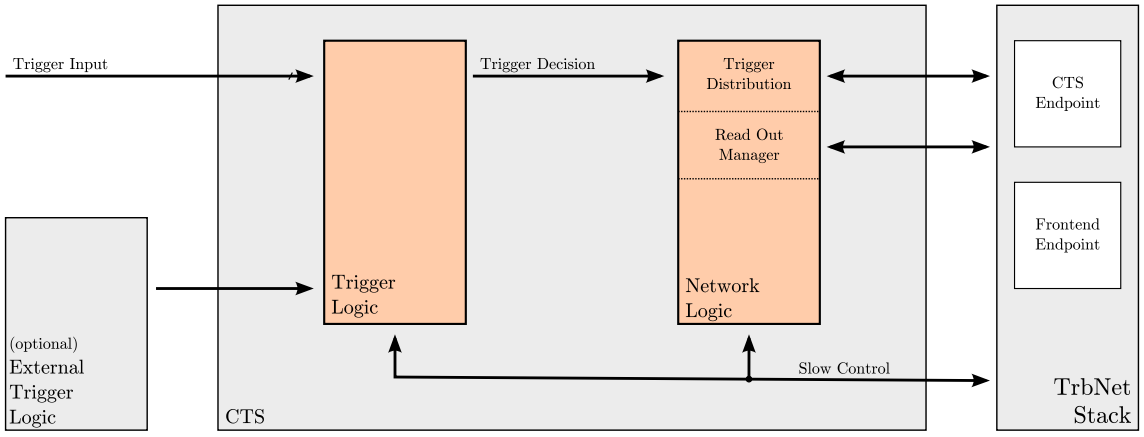

Figure 4: Structural overview of the CTS and its building blocks. The Trigger Logic (see chapter 3.1 and figure 5) monitors the inputs and makes a trigger decision. It is propagated to the Network Logic (section 3.2 and figure 10) which interacts with the TrbNet stack.

3 The CTS

From the user's point of view, the CTS is the pacemaker of a TrbNet-based DAQ system. It is a logical device, uniquely present in the network and coordinating both the capturing and the read-out of data. This chapter describes its structure and features and discusses some of the decisions made whilst its design.

As already discussed, the primary hardware platform is the TRB3 (see chapter 2.4), but no special functions of the FPGA are used by the CTS's logic. Therefore any TrbNet-capable board with sufficient logic blocks should be able to house the CTS.

In fact, a TRB2-board with a CTS add-on was used for the early development stages – mainly, because the required network endpoints already existed, but not less importantly, because the synthesis for the smaller ECP2 runs about twice as fast, thus increasing the development and debugging speed.

In order to increase the hardware independence of the design, the CTS consists of two major building blocks: The *Network Logic* handles the network interfaces and propagates event information gathered by the *Trigger Logic* (see figure **??** In theory, this strict encapsulation not only allows for a flexible trigger logic, but also permits replacing the network stack by any other compatible protocol.

3.1 Trigger Logic

3.1.1 Keep things manageable - a fundamental design decision

The *Trigger Logic* is the bridge between the generic network logic and the signals from the experiment. As the CTS is not designed for a specific setup, there are little restrictions on the nature of those signals.

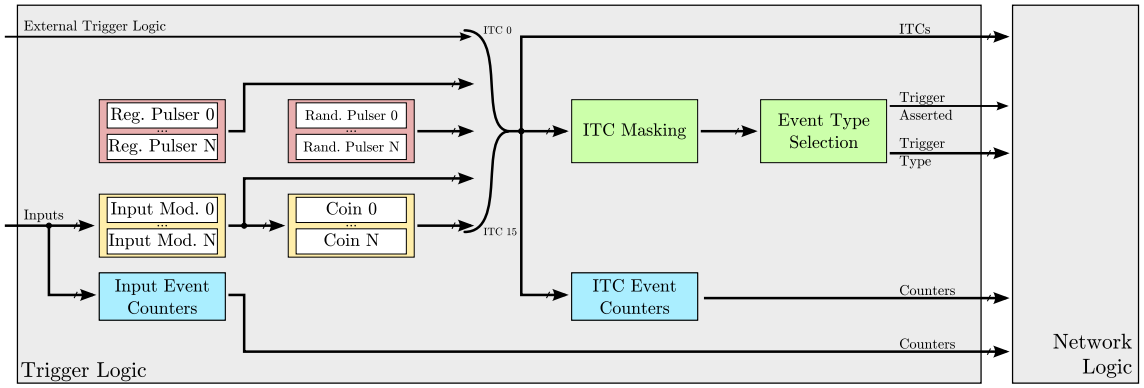

Figure 5: Block diagram of the trigger logic. Each coloured block indicates a trigger module. The slow control logic is omitted. For details on the network logic see chapter 3.2 and figure 10.

In the simplest case possible, a single line is asserted every time an event occurs, e.g. if a particle hits a detector. Since virtually all sensors have a limited detection efficiency, important events can be missed. In many situations this approach is very susceptible to noise, caused by the detectors, the amplifiers or irradiated via the cabling. Under the assumption that these disturbances occur randomly, the usage of multiple channels is favourable as it cancels noise exponentially in the number of inputs (see section 2.3). A coincidence of hits from different sources is a typical implementation of that scheme.

A very different triggering method is required due to concrete plans to incorporate the TRB3 into an existing DAQ systems – for instance as a standalone multi-channel TDC. In such a case, the TrbNet is used for on-board communications only and the CTS has to work in slave mode, i.e. it monitors the foreign data stream and propagates events detected by another trigger logic. An example is the *WASA at COSY* detector in Jülich, Germany, which uses a 32 bit word based serial data communication to distribute the trigger information [20]. As the WASA event identification is incompatible with the TrbNet's approach, the TRB3's CTS generates its own id and includes WASA's id into the data sent to the event builder.

While plenty other scenarios are possible, these few examples sufficiently demonstrate that there is no easily usable trigger logic which is universal enough to support all setups. This dilemma can be mitigated by exploiting the fact that the CTS is merely a hardware description synthesised for an FPGA:

It is at least a waste of resources to target programmable logic with a complex trigger logic, that requires intensive configuration in order to work properly. It seems more plausible to use a modular and extensible trigger structure, that permits a simple and manageable core logic which should cover most needs. If additional functionality is required, it can be added using the same sophisticated tools employed to develop the CTS.

3.1.2 Internal Trigger Channels

The trigger logic internally offers 16 channels, subsequently often referred to as ITCs[20], to which modules are connected to. These channels in combination with the memory structure discussed in chapter 3.3 are the key concept to an extensible logic. Together both techniques allow to design universal modules with little code overhead. Furthermore, it is possible to include modules on a need-only basis, i.e. only the functions required by a specific setup are synthesised.

To prevent misfiring shortly after start-up, all ITC are disabled by default and have to be enabled via slow control. Each ITC can be configured to be sensitive to either rising edges or high levels. If an enabled channel is active, the trigger module propagates this information to the network logic.

The TrbNet assigns a trigger type to each event which controls how an endpoint reacts and which data is sent to the event builder. For instance, in addition to the default *physics trigger* usually caused by a measured reaction there typically are regular artificial events which instruct all instruments to send a status information. The trigger types can be set channel-wise during runtime. The type of a given event is then defined by the lowest ITC that fired.

By design, three different sources can lead to a trigger event:

1. Input signals driven by off-board electronics, such as a detector.
2. External Trigger Logic that does not belong to the CTS core trigger logic. For instance, logic used to decode a data stream from a different DAQ infrastructure.
3. Pulsers, i.e. units that artificially generate events.

3.1.3 Input module

Each input signal of the trigger logic is preprocessed by an independent *input module* to compensate typical issues of signals from off-board electronics, such as twisted differential pairs, improper relative signal runtime and electrical noise. Figure 6 illustrates the unit's structure.

While noise is mainly a problem of analogue circuits, it may affect digital systems as well, especially on long wires which can act as antennas. A proper hardware design minimises its effects. But practically all systems suffer from the phenomenon up to a certain degree. A typical symptom of digital noise are short level changes of signals that are meant to be constant.

[20] **I**nternal **T**rigger **C**hannel

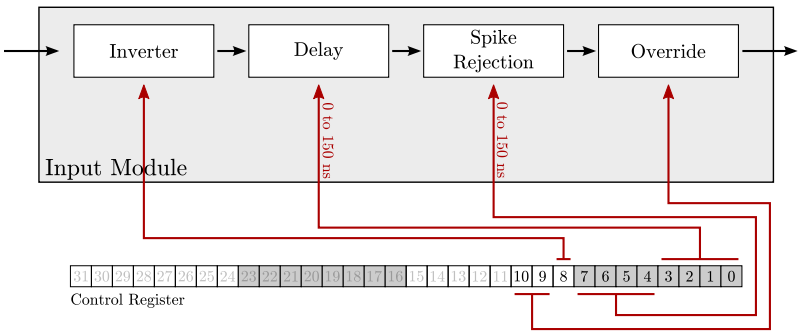

Figure 6: Block diagram of an input module. For each input signal an instance of an input module is generated. Its main task is to cancel out noise and equalise signals from different sources.

When sampling based on an edge triggered flip-flop with a frequency slower than the average spike length, it is very likely that the disturbance is either completely cancelled or visible for exactly one clock cycle. Thus, if the detector can be configured to generate pulses with a length of a few clock cycles, shorter pulses can safely be dropped, leading to a higher noise immunity.

The spike rejection logic can be used to dismiss pulses up to 15 cycles in length. It is implemented using a 4 bit counter that is incremented in each cycle while the input is high and is reset if the signal becomes low. As soon as the value exceeds the configurable threshold T, the pulse is considered valid and is propagated. This design introduces a delay of T cycles.

While normally all inputs should have the same spike rejection factor, the logic shifts signals relative to each other when different values are used. This might lead to problems – e.g. for the coincidence detection or in the later data analysis. A simple and computationally cheap countermeasure is to artificially delay the signals that traverse the rejection logic faster. However, as it is expected that those issues will rarely occur, a minimum latency is favoured and thus the additional delays are not introduced automatically.

Other sources of signal runtime are the limited speed of particles and secondary charge carriers within detectors, external circuits, such as amplifiers, and the limited velocity of propagation of the signal through wires and optical fibres (typically 2 m per clock cycle).

Independent of the origin, signals that are out of phase can manually be synchronised by delay lines. In an input module, a delay line is built from a 15 bit shift register and a multiplexer used to select the required delay.

3.1.4 Coincidence detection

The coincidence detection logic is used to detect rising edges and high levels of multiple signals within an adjustable window of time. It is expected that the unit is most commonly employed to cancel out statistical effects, such as noise. However, in combination with a (possibly external) delay line, the module can also detect a sequence of pulses.

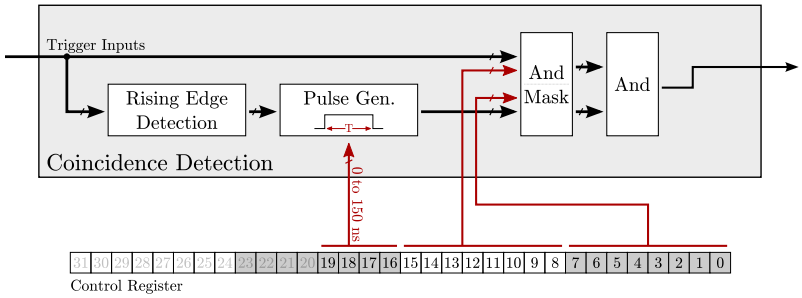

Figure 7: Block diagram of the coincidence detection

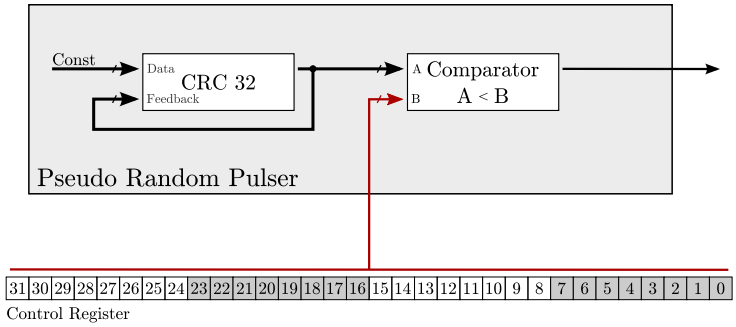

Figure 8: Block diagram of the pseudorandom pulser

The implementation is encapsulated into an entity which can be instantiated multiple times during synthesis. The number is limited only by the amount of free ITCs. Each unit can be configured individually: One bitmask selects a set of trigger inputs that have to rise within a configurable time window. For each input the logic internally generates artificial pulses that start with a rising edge of the signal and last for the coincidence time. Hence, as soon and long as the artificial signals of all selected inputs are asserted, the first coincidence condition is fulfilled.

While the former logic monitors changes of the inputs, there is a second bitmask used for level-sensitive conditions. The mask defines inputs that have to be asserted in order to propagate a edge-coincidence detected by the previous stage. These signals are called inhibit inputs as they can be used to filter events based on the state of an external low-active circuitry. If only one of the masks is the selected, the unit can be used to monitor asserted or rising lines exclusively.

3.1.5 Pulsers

Any module which leads to trigger decisions that are based on no input but the system's clock is considered a pulser. There are two pulser types implemented: A *random pulser* and a *periodical pulser*. Both are useful to schedule events, such as debug, calibration and synchronisation triggers, and allow for (stress) tests of the whole DAQ system.

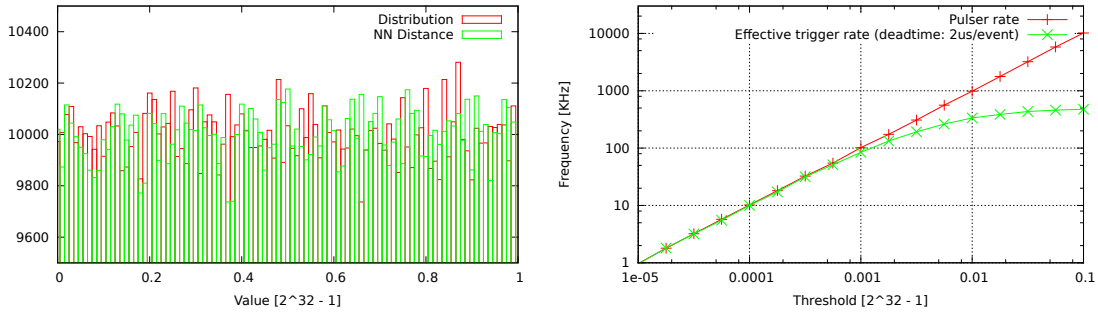

Figure 9: (a) Left: Normalised distribution of the pseudorandom number generator and the distance between two numbers of the sequence based on 10^6 iterations. A similar picture arises with less iterations, hence the uniform distribution seems to be no averaging effect caused by the large sample size.
(b) Right: Relationship between the threshold t and the rate f produced by the pulser. The simulation is consistent with the expectation $f(t \ll t_{\max}) = \frac{t}{t_{\max}} f_{\text{system}} = 100\ \text{MHz}\ \frac{t}{t_{\max}}$, which exploits the uniform distribution. As the pulser's rate approaches the maximal trigger rate, a growing number of events are missed. Hence the asymptotic behaviour.

A periodical pulser repeatedly asserts its output, followed by a configurable pause. The interval can be specified with a resolution of 10 ns ranging from a continuously asserted signal to one event per 42.9 s [21]. As the network is able to process events only up to a (system dependent) rate well below 1 MHz, the highest usable duty cycle is about 1%.

At first sight, it therefore is plausible to add a prescaler, which divides the pulser's clock frequency and hence allows for even longer intervals. The reason not to do so is the reciprocal relationship between the period time and the resulting frequency, that causes the smallest difference between two selectable rates to grow as the maximum frequency is approached.

Thus, by offering a maximal rate much higher than the utmost sampling rate of the DAQ, a comparably fine granularity for the actual usable rates is achieved.

While period pulsers are useful for reoccurring tasks, they may be inappropriate for the simulation of real – statistically distributed – events, for instance during stress tests. In these cases, a random pulser is a better choice.

The implementation available in the CTS employs a 32 bit CRC[22] unit with a fixed data word at its input to generate pseudo random numbers (PRN). As simulations suggest, the values generated are nearly uniform deviates. Furthermore, the distance between two

[21] The interval is limited by the 32 bit counter which produces at most 2^{32} before an overflow occurs and the clock rate of 100 MHz.

[22] **C**yclic **R**edundancy **C**heck. Intended for the verification of an arbitrary data stream, it computes a checksum based on the current data and the last result using a binary polynomial

successive numbers is also distributed almost uniformly and seems to be uncorrelated to their magnitude (see figure 9).

In each clock cycle a random number is compared with a configurable threshold. If it is smaller, an event is produced. Since the numbers are uniform deviates, the average duty cycle of the pulser is given by the threshold divided by the maximum value possible (see figure 9a). In addition, the uniformly distributed distances prevent the clustering of events that can be observed with other pseudorandom number generators, such as linear feedback shift registers.

In the case of unwanted grouping, multiple events occur in a short window of time and are followed by comparably long pauses. Because of the dead time of network, only the first event of the cluster is detected while the other ones are dropped. Hence, the long pauses dominate the effective rate which can be orders of magnitude smaller than the pulser's actual average rate.

3.1.6 External Trigger Logic

One use case of external trigger logic arises from the TRB3's universal design. The combination of a high amount of IOs and the comparably large number of programmable logic cells renders it an attractive add-on or upgrade for existing experiments. In such a setup the board is triggered by a network other than the TrbNet. Thus, an endpoint, i.e. a network interface, compatible with the foreign protocol has to be included in the central FPGA's design.

The network adapter naturally has at least two interfaces – one interacting with the foreign network, the other one with the CTS. As one can hardly make any assumptions about the physical layer of the master DAQ, a structure to cope with different interface designs is required. There are may differences regarding the bus width (serial vs. parallel), the handshaking or clock handling. Due to VHDL's limited support for dynamic port, it seems plausible not to include the endpoint into the CTS, thus allowing for a fixed CTS port. The same argument applies to most other applications of external trigger logic. Currently the CTS can operate as a client for the MBS protocol[23].

In contrast to common trigger modules which are instantiated within the trigger logic's architecture and therefore inside the CTS's component hierarchy, *External Trigger Logic* lays outside – typically on the same level as the CTS's main entity.

[23] Decoder implemented by Jan Michel

The interface to the external logic consists of:

- A `generic`[24] used during the synthesis to communicate the module's type-id (chapter 3.3.1).
- A direct link to the ITC with the highest priority.
- A busy signal from the CTS which might be used by an adapter for a foreign network that implements a busy-release scheme.
- A dedicated readout interface enabling the module to send data to an event builder during the readout process.
- One status and one control register managed by the trigger logic. If more slow control registers are required, the top entity's bus handler can be used.

3.1.7 Options to increase the trigger's accuracy

In the current implementation, the whole CTS is driven by a single clock running at 100 MHz. There are two important reasons for this approach. Firstly, it makes the design more universal as it minimises the requirements on the hardware – the 100 MHz clock is also needed by the TrbNet endpoints and hence can be taken for granted. Secondly, as each interface between clock domains introduces a certain overhead, the usage of a single time basis simplifies the system.

Nevertheless, it is worth analysing how this design affects the system's performance and which subsystems of the CTS could benefit from a higher clock frequency. Without additional hardware, the fastest time base available on the TRB3 clocks at 200 MHz, i.e. twice as fast as the one currently used. An ECP3 PLL can further increase this frequency to up to 400 MHz.

The main advantage of a higher clock frequency is the increased temporal resolution of the trigger logic. It is, however, hardly possible to migrate the complete trigger module into the faster clock domain. Especially the counters used in the statistics and pulser modules have long critical paths which limit their maximum speed. However, both units should have a sufficient accuracy.

The input- and coincidence units are more qualified for an upgrade. While both have – by design – a structure suited for high sampling rates, some experiments may actually utilise a finer granularity of the spike rejection or delay compensation as well as a sharper

[24] special type of constant in VHDL

coincidence window. The faster domain is already provisioned in the code. It only requires an exchange of the modules' clock signals and an additional set of flip-flops at the corresponding ITCs to counteract the meta-stability at the domains' border.

Another doubling of the sample rate can be achieved by using the double-data-rate approach, which samples at the rising and at the falling edge of the clock signal, but this is far more involved and requires major adjustments.

For setups that require the exact arrival time of an event, it might be interesting to adopt the TDC[25] design implemented for the peripheral FPGAs of the TRB3. Using one channel per input and an additional channel for the system clock as reference, the precision of the temporal information can be increased by three orders of magnitude.

3.2 CTS Network Logic

3.2.1 State of the Art

Before discussing the CTS Network Logic it is helpful to further introduce the structure and semantics of the TrbNet. This can easily be done using the concept of *virtual machines*.

Virtual machines are commonly employed to simplify the design of a complex system. By using this scheme the whole system can be visualised as a stack of abstraction levels. Each level is represented by a black box which offers a certain service to the higher level by using the functionality provided by the lower one. It is possible to implement different approaches to a problem and subsequently choose the best one for a given application.

The ISO/OSI [22] is a standard reference model based on this design method for any sort of computer network. In total, it defines seven layers, ranging from the physical layer, which implements the bitwise data transfer, up to the application layer defining a high-level protocol for a specific task.

When matching the TrbNet's implementation against this model, some machines are merged and four levels remain. The lowest layer, referred to as the *media interface*, handles the data transfer and link functionality. Because of its hardware dependence there are multiple implementations differing in the transfer technology[26] used and the requirements for the FPGA.

The next level implements the TrbNet's basis responsible for packet routing and session handling. It is not discussed further (see [17] for details). The following machine is often referred to as the endpoint's logic. There are two endpoint types relevant for the CTS:

[25] Time to Digital Converter with a binning size less than 14 ps RMS [16]

[26] e.g. optical transceivers based on SFP modules, FOT, on-board communications between multiple chips, or on-chip as used for the CTS implementation for the TRB3

- The *CTS endpoint* offers two simple ports to distribute trigger information and to issue readouts. In order to send a request, the application only has to provide the event identification information accompanied by a strobe signal. The endpoint then becomes busy until it receives an acknowledgement from all connected devices or a timeout occurs. To minimise the dead time of the system, both functions work asynchronously.

- The *frontend endpoint*, also referred to as FEE or data endpoint, implements the complementary features. If it receives a trigger request, it informs the application logic which is supposed to execute the measurements and write the response into a FIFO buffer managed by the endpoint. Only then the afore-mentioned acknowledgement packet, serving as a busy release, is sent. The readout process is completely handled by the endpoint's logic without any interaction of the application's logic.

The TrbNet's specification requires a buffer able to store at least two complete events, but usually it is designed to hold approximately 200 events. While the FIFO offers no benefits in scenarios with a fixed and regular trigger rate, it actually reduces the number of lost events in most particle physics experiments. This is due to the common phenomenon of bunching in the beam and the statistical distribution of the events which both lead to spikes in the reaction rate. The delayed readout helps to level out those short term fluctuations as it leads to a smoother data rate on the network.

3.2.2 Structural Overview

The CTS uses two dedicated network endpoints for communications – one of each type described in the previous section. The *CTS Endpoint* is obviously needed due to its unique ability to send trigger packets and coordinate the readout process. However, it lacks the support to transmit arbitrary data to the storage servers. There are two main reasons for this design decision which was made independently from this work: Most importantly, it allows for a simple and more consistent data flow to the network. Secondly, it is easier to instantiate the already existing FEE code than to adopt and separately maintain the functionality for the *CTS Endpoint*.

The TRB2 board, used in the early development stages, actually required two complete instances of the TrbNet stack and two independent optical links to a remote hub. The TRB3 board, on the other hand, already uses a hub in the central FPGA to communicate with the peripheral chips. Thus, it appears advisable to embed the CTS into the former FPGA and thereby to eliminate the need for four media interfaces (two for the CTS, two in the hub).

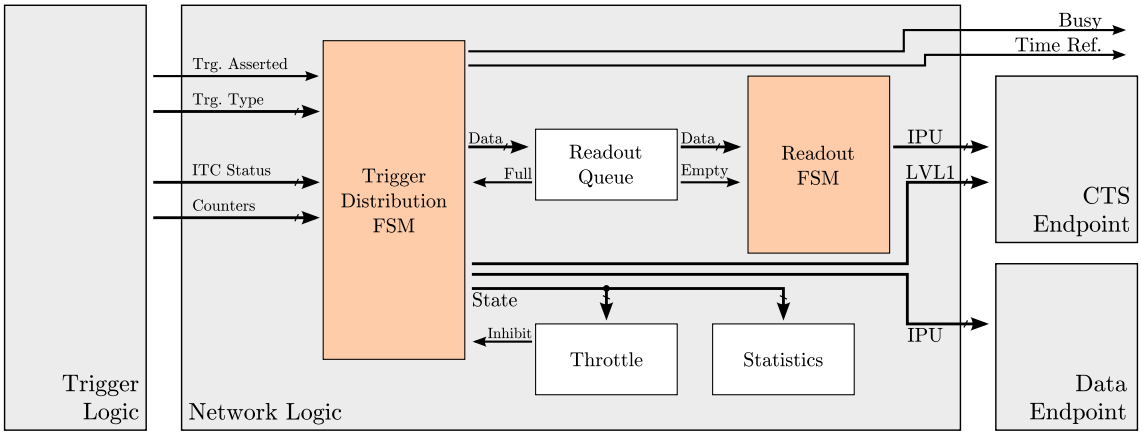

Figure 10: Schematic overview of the CTS Network Logic

The hub and two endpoints are encapsulated into a dedicated TrbNet component[27] and are instantiated independently from the CTS in the top-entity of the design. Due to this strict encapsulation, the CTS's code is identical for both boards.

As depicted in figure 10, the CTS's main functionality is modelled by two connected FSMDs.[28] Both automata have a very linear behaviour, i.e. the individual states are primarily used to schedule the programs' outputs over time and not to implement alternative flow paths. Each machine starts and remains in an idle state until a start condition is met. As soon as the machine becomes busy, a fixed cycle of actions is started, which eventually reaches the initial idle state again. Only the holding time of each state varies – depending on the network's status and the CTS's configuration.

3.2.3 The state machines

The *Trigger Distribution FSM*, often referred to as *TD-FSM*, is much more complex than the second automaton and has the following program sequence:

1. The **idle** stage waits until the trigger logic detects an event of interest. There are a number of inhibiting factors, such as a full *readout queue*, a throttling mechanism used to limit the trigger rate or debugging tools. Upon transitioning to the next state, a snapshot of all the values that are later sent to the endpoint's buffer is latched.

2. The **trigger distribution** generates the trigger identification information, i.e. a 16 bit sequential number and 8 bit pseudorandom number, and issues the CTS endpoint to transmit this information. If required by the trigger type, an additional 100 ns time reference signal is generated.

[27] implemented by Jan Michel and not subject to this work.

[28] Finite-state machine with datapath, i.e. a FSM which controls a feedback datapath

While there are many pseudorandom number generation schemes available, the generator employed was chosen because of its simple implementation. It computes the sequence

$$x_{i+1} \cong x_i + 113 \mod 256,$$

which has a period of 256 and hence is uniformly distributed. The high correlation of successive values is not relevant as the independently working FSM only rarely picks samples. Thus, the (often statistically distributed) sampling time implicitly acts as a source of entropy.

3. The **data transmit phase** starts when the trigger information sent in the previous stage arrives at the CTS's frontend endpoint. The latency strongly depends on the hardware used. For the TRB3 – where CTS and hub are on the same chip – it is dominated by the hub's logic and causes a delay of approximately 400 ns for a round-trip. This is about twice as fast as the round-trip latency required for the serial communications with the peripheral chips [18]. Thus, as long as a regular data package contains less than 40 words, the CTS introduces no additional dead time during data transmission.

   While this should always be the case, the exact amount of data depends on the CTS's setup. The packet therefore contains a header word identifying its structure. The following information can be included:

   - The number of rising edges and cycles asserted of each trigger input
   - The number of rising edges and cycles asserted of each active ITC
   - The last idle- and dead time of the CTS
   - The number of detected events and the number of triggers accepted

   All these values are useful in the later data analysis. For instance, they allow to derive scaling parameters to generalise the events observed to the actually occurred reactions. A complete packet contains more than 200 byte raw data. At a trigger rate of 100 KHz and neglecting protocol overhead, this sums up to roughly 50 % of the bandwidth of an ethernet uplink and therefore is prohibitively expensive.

   However, as each counter overflows at most once per 42.9 s and because most values are accumulated, it is not necessary to include all data in each event. The same applies for most status information of the other devices present in the network.

   Thus, the CTS can and should be configured during runtime to include a minimal amount of data in regular events. An additionally scheduled slow *debug trigger* (e.g. with 1 Hz) can be used for a complete and system-wide dump.

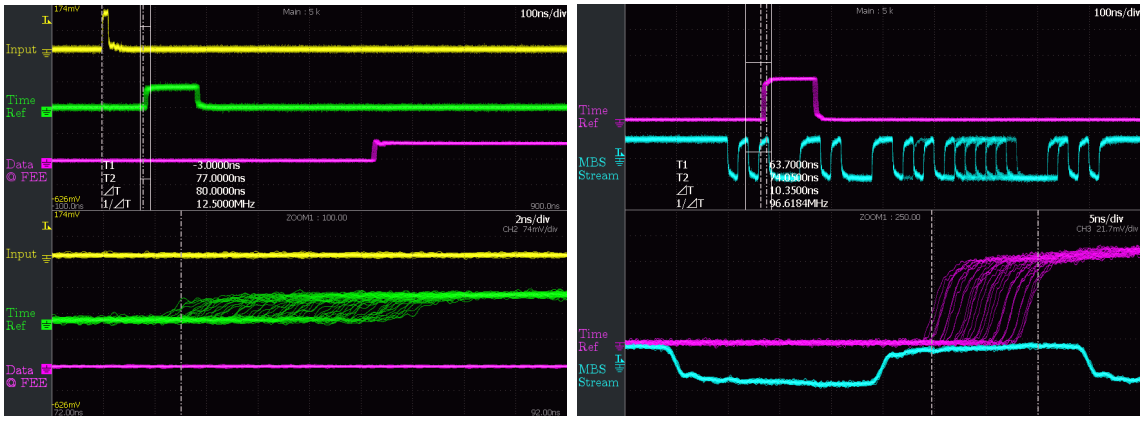

Figure 11: Latency and jitter measurement ... **(a) Left:** ... using trigger input. Yellow: Input, Green: Time Reference signal, Purple: Arrival at frontend (on board) **(b) Right:** ... using MBS module. Aqua (blue): MBS data stream (input), Purple: Time Reference. The first falling edge of the MBS data stream starts the trigger message.

The only information that is valid exclusively for the last event are the idle- and dead times. However, it is expected that these values are mainly used for debugging purposes as the average times should be sufficient for most experiments.

4. When the busy releases from all frontends are received, the **readout process**, i.e. the data transfer from the FEEs' buffers to an event builder, can be started. While it is possible to issue this request with the TD-FSM, doing so would unnecessarily increase the dead time of the system. Therefore the last stage of the machine only pushes a token containing all trigger identification information into the readout queue and then immediately becomes ready to accept a new event.

The second state machine, responsible exclusively for the readout process, becomes busy as soon as at least one token is in the readout queue. In this case, it issues the readout request via the CTS endpoint and waits until the endpoint becomes idle. It then dequeues the token and starts over.

As a token consists of one 32 bit word only, it is small in comparison to the average data packet stored in a frontend's buffer. It is therefore reasonably cheap to choose a queue size that is able to hold more events than the smallest FEE FIFO can. The easiest way to implement such a queue efficiently is using two-port embedded RAM blocks whose capacity, in turn, suggests possible queue sizes. In normal operation mode a single 18 kBit block, able to store 512 tokens, should be sufficient.

3.2.4 Latency and Jitter

An important measure of a trigger system is the delay introduced between the arrival of an event and the distribution of the trigger decision. Due to the CTS modular structure,

this quantity is strongly influenced by the actual trigger modules used. Figure 11 includes the measurements for two different channels, both recorded on a TRB3:

- The setup of the first plot directly uses an external trigger input and. Thus, the signal takes the following path: In order to avoid meta-stabilities, the input is sampled using a flip-flop and then routed to an input module, which takes 3 cycles for the shortest delay and spike rejection settings. The propagation delay grows linearly, if either of those values is increased. The ITC handling further requires 2 cycle, followed by additional 2 cycles until the time reference signal is available. In total, this sums up to a delay of 80 ns. On a TRB3 board, the TrbNet stack requires another 450 ns to deliver the LVL1 packet. While this number significantly influences the system's dead time, frontends with critical timing constraints typically use the time reference.
- The MBS input module is an external trigger module sampling a serial data stream of 50 Mbits^{-1}. It propagates the trigger information as soon as the specified start-pattern is received, resulting in an overall delay from the first falling edge of the start packet to the rising edge of the time reference signal of 6 clock cycles.

Both latencies measured have a positive jitter of 10 ns, i.e. the sampling period of the CTS running at 100 MHz. Thus, the design itself can be considered jitter free. As discussed in section 3.1.7, the sampling frequency is a systematic upper bound of the temporal resolution of any sequential circuit. In order to quantise the discrepancy between the arrival of an event and the sampling time for a given instance, a TDC can be used to measure that offset. While that quantity can be used for corrections during the off-line data analysis, for the actual triggering process, the jitter remains.

3.3 Slow Control

The TrbNet supports slow control, i.e. a generic scheme to access properties and status information of a device via a register-based approach. Most registers are handled by the application's logic which is connected to the endpoint by a 32 bit parallel data bus with a 16 bit address lane and a few handshaking signals, such as read- and write- strobes and acknowledgement bits. While the bus implements a point-to-point connection, cascadable bus handlers can be used as address-based multiplexers in order to connect multiple clients.

There are a number of common registers that are controlled by the endpoint itself; other address ranges are reserved for network devices, such as hubs. As the HADES CTS uses addresses starting at `0xa000` and because multiple CTSs are not sensible within the same endpoint, this range is used without the risk of collisions.

Just like the CTS's structure, the address space itself is separated into two major blocks, one controlled by the *network logic* and the other one by the *trigger logic*. As the former is not likely to change substantially in future developments, the first block has a fixed register layout shown in table 4.

The address block assigned to the *trigger logic* has to be more flexible as extensions to the trigger are very likely and encouraged by its design. Therefore a protocol is required that allows remote software to automatically identify and access all modules included in a specific build.

One solution to this problem assigns fixed addresses to each module. In this case the software can determine whether a certain feature is available by reading from any related register. If the module is not included in the hardware and thus the address unmanaged, the endpoint returns an error code which signals the module's absence to the software. However, this approach requires significant administrative efforts as all registers need to be documented and must be available to every software interacting with the CTS. More severely, if a module requires additional registers, the address space may become fragmented which considerably increases the access times.

3.3.1 A flexible address scheme

To avoid those issues, a more flexible and powerful approach is used: Each module specifies its own – by definition – continuous address layout relative to a base address. During synthesis all individual blocks available are joint together, connected only by a header as described in table 3. The header identifies the following block by an 8 bit ID that includes the block's size and further informs the software which trigger channels are connected to the module.

Thus, when initially connecting to the CTS, the software has to read the first header from the address `0xa100`. Even if the block's ID is unknown, the next header's address can be derived using the length information included in the header word. The enumeration is completed when the last header – indicated by its highest bit – is read. The order in which the modules appear depends on the specific hardware description, but it is not defined by any convention.

The only restriction is that every module id appears only once. If the trigger logic contains multiple instances of the same module, they have to share a block, which can be indicated by the header's length information. This decision reduces the amount of header words and thus speeds up the enumeration process. It further decreases the code complexity of the client software.

Bit(s)	Description
	Block identification header
7:0	Block type
15: 8	Number of addresses in this block excluding this header word
20:16	First internal trigger channel assigned to this block (0 if it does not apply)
25:21	Number of internal trigger channel assigned to this block (0 if it does not apply)
31	Last block indicator. Enumeration stops after reading this block

Table 3: Header used to identify an address block within the trigger logic's address range

Currently the following IDs are used:

- `0x00` **Internal Channel Masking**. This block contains one control register holding two bitmasks. Each of the lower 16 bits enables one ITC, while the upper 2 bytes select whether the channel is edge- or level sensitive. After a reset all channels are disabled to ensure that no trigger is distributed before the whole network is initialised.

- `0x01` **Internal Channel Event Counter**. This block contains two 32 bit counters for each of the 16 ITCs. The first word of every pair represents the number of clocks in which the trigger channel was asserted, the second one holds the number of rising edges. All counters work independently and overflow without any notice. They have to be polled at least every other 40 s to ensure that no register overflowed more than once.

- `0x10` **Input Module Configuration**. Each register holds the configuration of one input module as discussed in chapter 3.1.3.

- `0x11` **Input Event Counter**. This block has the same structure as `0x01`, however, its counters monitor the trigger inputs before they are processed by the input modules. Hence by comparing both counter types, one can infer the number of events filtered by the spike rejection.

- `0x20` **Coincidence Configuration**. Each coincidence detection module (see 3.1.4) has one configuration register. Thus, the number of registers inside this block matches the number of `COIN`s.

- `0x30` **Periodical Pulser**. Each register in this block stores the low-period's length of a pulser in clock cycles. 0 results in a constant high channel.

- `0x40` **Event types**. This block contains exactly two registers that assign a *trigger type ID* (4 bit) to each ITC. Starting with the type of the first channel at the first register's lowest nibble, each word stores 8 types.

- `0x50` **Random Pulser**. A random pulser generates irregular event patterns. Each instance is configured with one control register, which holds its threshold. As discussed in chapter 3.1.5, there is a linear dependency between the average trigger rate F and the threshold T given by $F(T) = \frac{100\ \mathrm{MHz}}{2^{32}-1} \cdot T$.

- `0x60` **External logic- CBM/MBS**. This module indicates the presence of the CBM adapter module. If set, the lowest bit of the control registers prevents the module from sending data to the event builder. The lower 24 bit of the status register contains the timestamp of the last event seen. The MSB holds the error flag.

It can be argued that the usage of two distinct address ranges is inconsistent with the dynamic address scheme available. Nevertheless, there are a number of reasons for this decision: Firstly it introduces only minor drawbacks, such as a small fragmentation. It is therefore not possible to read all CTS registers with one request.

Secondly and most importantly, the two ranges are consistent with the separation of the two building blocks: Keeping the functionality apart while using a common slow control logic seems implausible and complicates the structure of the hardware description by introducing additional dependencies.

4 Software Tools

The software created for the CTS is designed to assist the user with the configuration and the monitoring of the device. It is operated from a remote PC and communicates with the CTS via the TrbNet's slow control features (see chapters 3.3, 2.2). There are two user interfaces based on a common application logic:

- The **low-level interface** (CLI[29]) will most likely be primarily invoked by external scripts. It allows to display, manipulate, and export *named* properties of the CTS in a human readable fashion.
- The **graphical user interface** (GUI[30]) is based on an interactive web service. It presents settings and status information in a semantically structured way and offers real-time rate plots.

This chapter introduces the tools created. Starting at the low-level functions, each section describes a new abstraction layer. Furthermore, some aspects of the techniques and programming languages that affect the software's structure, are discussed.

4.1 The Foundation

The direct interaction between software and hardware has typical problems and common patterns to approach them. In the hardware, for instance, many registers contain several independent values. While this can reduce the hardware consumption, in software, it is often more convenient to extract the different components and cope with them individually. This requires repeated logical operations, such as bit- masking and shifting, which are not easy to maintain and prone to hard detectable errors when programmed by hand.

A small framework was developed to address these issues. It builds on top of the existing TrbNet bindings and can be used to communicate with any endpoint in the network. Based on an address layout description provided by the application it simplifies the access to the remote device's registers and automatically performs boundary checks. Figure 12 depicts an overview of the foundation classes.

While the software basis, such as the `libtrbnet` library or the `trbcmd` and `trbflash` tools, are implemented in C, Perl is the dominating programming language for high level functions in TrbNet related projects. For that reason, most parts of the software created in this work use this language. Only the functionality of the GUI web service, which is executed within the user's browser, had to be implemented in JavaScript (see section 4.2).

[29] **C**ommand-**L**ine **I**nterface

[30] **G**raphical **U**ser **I**nterface

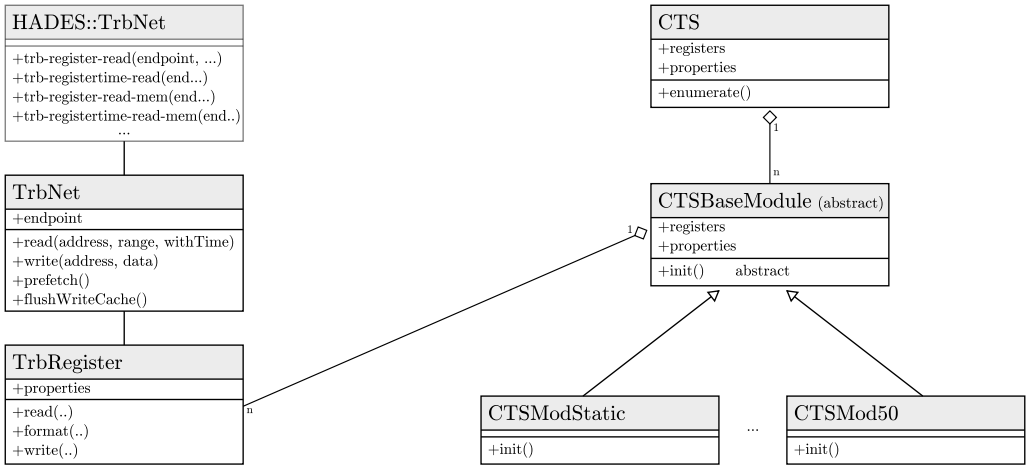

Figure 12: Simplified class diagram of classes belonging to the tool foundation. Many methods and attributes are omitted. The `HADES::TrbNet` block represents a procedural package.

Perl[31] is a widespread, free, and open-source scripting language. In contrast to a program written in a compiling programming language (such as C, C++ or Fortran), a Perl script is compiled and optimised each time the program is started. It then gets executed within a virtual machine. A number of common programming paradigms are supported. Scripts, for instance, can have aspects of procedural, object-oriented, imperative or functional concepts.

The language itself has a core functionality that offers multiple basic data structures and operations on them, as well as functions to interact with the program's environment. Perl has a dynamic, implicit and weak type handling and uses variable prefixes, so-called sigils, and special operators to express how data has to be interpreted. Additional functions are loaded via modules that can be implemented in Perl or C. The existing `HADES::TrbNet` library is an example of a module implemented in C, while all developments introduced in the following are pure Perl scripts.

4.1.1 Slow control interface

The TrbNet bindings for Perl offer a procedural interface that defines functions to issue read and write commands to single registers and continuous memory ranges. As the plug-in reads the session's environment variables, no explicit configuration from the script is necessary. Solely an endpoint and parameters, such as the register's address, have to be provided to invoke a slow-control command.

[31] In the interest of improving readability, the version number is omitted. In this work Perl 5.12+ was used. All statements refer to this release.

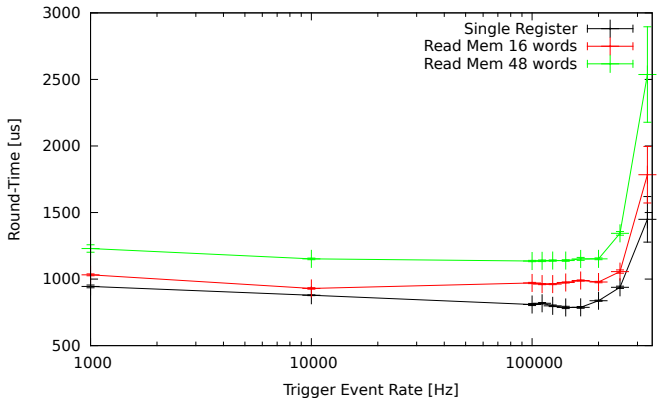

Figure 13: Latency of read-accesses via slow control under different network loads. The absolute values strongly depend on the system used (data was gathered with a Perl script using the TrbNet Perl bindings. A Trb2 / Etrax board was used as a TrbNet bridge. Each point is averaged over 1000 measurements) It is obvious that the run-time is dominated by constant latencies (introduced primarily by the UDP/IP network stacks). The amount of data read, in turn, only has a small effect on the round-trip time. Hence block reads are favourable.

On read-access, it is possible to request an additional timestamp. It is appended by the remote endpoint and indicates the access time with a resolution of 16 us. This feature is used to increase the accuracy of rate measurements (see chapter 4.1.4).

The package uses a synchronous communication model, i.e. once a request is invoked the application blocks until a response is received or a time-out occurs. In contrast to an asynchronous approach, such as an event-driven program structure, a blocking access scheme often simplifies automated scripts. In order to guarantee a fast and reactive user interface, it is, however, advisable to use another (non blocking) thread or process to handle the user interface. The CTS' GUI is implemented in this way, albeit there are more important reasons for this design (see chapter 4.2.1).

A new abstraction layer based on the described interface is used to offer additional functionality. Initially, the layer was used to allow for unit and integration tests on machines without the TrbNet bindings or without a connection to the CTS hardware. In this operation mode, a previously recorded memory dump simulates the initial state of the remote device. Even if the code is still present, it is not used by the application itself.

The new interface is object-oriented and represents each endpoint with an instance of the class `TrbNet`. It offers two methods, `read` and `write`, to issue any access described earlier. Parametric polymorphism is employed to select the access mode.

As figure 13 suggests, block reads are significantly faster compared to repeated single register reads. To exploit that fact, the class supports a semi-transparent read cache. Due to its synchronous access model, it is not possible to automatically delay multiple reads in order to combine them.

```
$regs->{'cts_ro_queue'} = TrbRegister->new(0x07 + $block, $trb, {
   'count' => {'lower' => 0,  'len' => 16},
   'state' => {'lower' => 30, 'len' =>  2, 'type' => 'enum', 'enum' => {
      0x0 => 'Active',
      0x1 => 'Empty',
      0x2 => 'Full'
   }}
}, {
   'accessmode' => 'ro',            # status register:  read-only access
   'monitor' => 1,                  # instructs monitor daemon the regulary
                                    # refetch the status
   'label' => "RO Queue State"      # Label used in CLI interface
});

print $regs->{'cts_ro_queue'}->format()->{'state'};
```

Listing 1: Example of the usage of the TrbRegister class. The instantiation code is directly taken from the CTS tool. It models the read-out queue's status register (see section 3.2.3). The lower two bytes represent the number of tokens stored in the fifo, while two upper bits hold the queue's internal state. The enum-type is used to decipher this information. The last line was added to demonstrate, how a given information can be read from the endpoint (or local cache) and formatted for the user.

Therefore, the application can provide a list of registers that are going to be read in the near future and subsequently instruct a prefetch. During prefetch all continuous registers are grouped together. If there is a small gap between two address ranges, the class decides based on a database of known registers whether the not requested space can be read safely. In this case, all three areas are merged and retrieved with one request. If the process is completed, the cache works transparently, i.e. the `read` method only accesses the TrbNet endpoint if the requested register was not prefetched. If necessary, the application has to clear the cache explicitly.

A second cache is used to accelerate write commands. Once activated, it works fully transparent and delays all write accesses until they are explicitly committed. On read accesses, the write cache exceeds the read cache's priority. An analysis of the access patterns shows that the CTS software only rarely manipulates values of two registers at a time. The write cache, therefore, is comparably small. It is nevertheless important as it also speeds up multiple writes to different bits of the same register. They commonly occur due to the sliced register concept introduced in the following section.

Because of the TrbNet's structure, cache coherence protocols cannot be implemented efficiently. Even if the local changes to the write cache are passed transparently to the read cache, race conditions can occur as soon as two independent processes alter the same register. As discussed later, this should not cause any severe problems to the CTS GUI.

4.1.2 Named and Typed Registers

The concept of named and typed registers aims to prevent code redundancy and ease the hardware access. It builds on top of the TrbNet abstraction described in the previous

section and is encapsulated into the class `TrbRegister`. When enumerating the synthesis configuration of the CTS, a class instance for each available register is created and registered in a central database with a unique *name*. Subsequently all scripts can (and should) use this identifier to address a register. This not only increases the readability of the code, it also eliminates almost all explicit address calculations that are otherwise necessary to adopt to the trigger logic's dynamic address scheme (see chapter 3.3.1).

Additionally to the address, type information of a register can be provided. In this scheme, the register is split into slices, i.e. a number of continuous sequences of related bits. Each slice is marked with a human readable identifier and has a type, that is used to restrict the slice's value space and influences how it is presented to the user.

The following types are supported:

- **uint**: The slice contains an unsigned integer that is displayed as a decimal number.
- **hex**: Same as uint, however, the value is printed out as a hexadecimal number with a `0x` prefix.
- **bitmask**: The slice holds a bitmask which is presented to the user as sequence of binary nibbles.
- **enum**: Similarly to an enum-type in many programming languages, one can define a descriptive name for a numeric value, e.g. to decode the state's binary representation of an FSM.

The class automatically performs a boundary check of a slice's value on each read and write access and reports errors to the user. Furthermore, each register object manages a set of properties. They control how different parts of the application deal with the register. The property `accessmode`, for instance, informs the framework whether the hardware supports read and/or write accesses to the given address. The attribute `const` prevents read accesses, as any read request on that address returns the attribute's value. This is especially useful if the value of a register is already known, e.g. if a trigger header was captured during an earlier enumeration. Listing 1 shows other examples of properties.

Experiments with another abstraction layer were conducted: An instance of `TrbSlicedRegister` manages a virtual register which is defined as a combination of slices from possibly multiple hardware registers. It does not have the size limitations that are imposed by the hardware's 32 bit architecture. Thus, it is possible to gather all properties of one module in a single virtual register. This concept was discontinued, because the CTS gives little justification for this additional overhead – especially as severe changes to the CTS's register mapping are not anticipated.

4.1.3 CTS low-level binding

In contrast to all generic features discussed so far, the CTS low-level bindings are designed to interact with the CTS exclusively. The package consists of several internal classes and the interface is implemented via the `Cts` object. It stores a central file containing the remote device's properties and registers and handles the enumeration process in which this information is gathered.

When connecting to a CTS, the headers of all trigger modules are read. The class then tries to find and dynamically load a plug-in file, which is identified by the header's id. The plug-in is supposed to create `TrbRegister` definitions for all registers of the trigger module and store them centrally in the CTS object. As every plug-in is a child of the same dedicated base class, almost all common code structures of this process are managed implicitly (see Appendix A.2 for an exemplary plug-in file). If no suitable driver is found, a warning is printed and the enumeration continues.

As a result of all techniques introduced, the low-level software support for a new trigger module can be added by creating a single plug-in file which defines its registers. By tagging registers with properties, such as `monitor` (listing 1 and next chapter), they automatically are used in all generic user interface operations, e.g. by the monitoring daemon described in the next chapter.

The console-level interface enables the outside world, i.e. a user or a third-party program, to interact with the functions offered by `Cts` or the objects created by it. The CLI has two modes of operation. In the **monitoring mode**, the program regularly fetches the values of all appropriately tagged registers and displays the information to the user. This process is further analysed in section 4.1.4. In the **command mode**, the program terminates after the requested operation is executed. There are four commands available in this mode:

- The *list command* outputs a table of all available registers, slices and the corresponding hardware addresses.
- The *export command* generates a file containing the values of all CTS control registers. It is stored as a `trbcmd` script. This feature is intended to ease the start-up process of the DAQ system. Once the CTS is configured, the settings can be exported directly into the DAQ startup config files [12].
- The *read* and *write commands* allow to access all CTS properties in a human readable way. Both commands support a register or slice-based addressing. If an `enum`-typed sliced is accessed, both its numerical and textual representation may be used (see listing 2).

```
> ./cts read trg_input_config0
Key               | Address | Value      | Slice     | Slice Value
------------------------------------------------------------------
trg_input_config0 | 0xa124  | 0x00000203 | delay     | 3
                                         | invert    | false
                                         | override  | to_low
                                         | spike_rej | 0

> ./cts write trg_input_config0.override=off, trg_input_config0.delay=0
Done.
```

Listing 2: Examples of read and write commands issued with the CLI

4.1.4 Monitoring the CTS

As already discussed, the CLI of the CTS tool offers a monitoring mode in which all important status and control registers are read regularly. The data can be presented either directly as a text-based table in the terminal or it can be written into several machine readable files. These, for instance, are used by the GUI (see section 4.2 for more details).

In a first step, the monitoring method gathers all registers with a `monitor` or `ratemonitor` tag and advises the TrbNet layer of the upcoming read requests. Using the prefetch scheme, most registers can be read within a block access. Depending on the modules included in the CTS, approximately 100 registers have to be read. The prefetch algorithm typically obtains their values with only two block requests. As suggested by extrapolation of figure 13 and experimentally confirmed, this takes significantly less than 10 ms under normal network conditions.

Registers with the `ratemonitor` tag are interpreted as counters. The rate is trivially computed as the difference quotient of the two most recent values and their timestamps. If the newer count is smaller than the previous, a single overflow is assumed. As all counters of the CTS run with at most 100 MHz, this is always true if the interval between the two samples is lower than 42.9 s.

Because of the unpredictable jitter introduced by multiple network stacks, the point in time of a given register access can be measured only with a resolution of a few milliseconds if based exclusively on the PC's internal clock. The TrbNet timestamp is used to increase the resolution to 16 μs. Its limited period of 1.048 s is compensated by the computer's system clock.

Finding a suitable sampling frequency for the monitoring process is influenced by a number of factors:

- In order to provide a reactive user interface, the data presented should be updated as frequently as possible.

- A lower sampling frequency reduces the network and computational load on all devices.

- A lower frequency increases the accuracy of the rate counts. There are two effects that can produce wrong or misleading counts. If the monitoring rate is similar to or higher than a measured rate, there are none to a few counts within a given interval. Therefore, each count has an increased relevance, which leads to so-called *digital noise*.

  The second effect is statistical in nature and arises from the limited time resolution of the measurements. As each rate measurement is based on two discrete timestamps with a given bin size, the interval length has an uncertainty. Its probability distribution has finite values in the interval of $]-16;16[\mu$s, peaks at 0 and linearly decreases to both sides. This convolution pyramid leads to a constant standard deviation of 16.5 μs.

  Both effects introduce a linear dependency of the sampling rate to the relative error.

As a compromise, once a sampling is completed, the monitor process requests the operating system to sleep for 800 ms. The rate, thus, adapts to the host's processor load and has an average of approximately 1 Hz. This results in relative errors below 1 ‰ for all measurements above 100 Hz.

4.2 Graphical User Interface

The console-level interface enables the user to administrate and monitor all features of the CTS. It, however, requires a significant knowledge about the tools and the CTS itself. Furthermore, the textual interface limits the possibilities of presenting the CTS' status semantically structured.

As depicted in figure 14, a graphical user interface offers more options to guide and assist the user, e.g. with plots or by grouping together related properties that are stored in different registers.

As the creation of a graphical user interface from scratch is a complex problem, there is a great variety of approaches. Microsoft Windows, for instance, defines a complete API[32] that allows to define windows and controls at a comparably high level. In this case, the operating system handles almost all aspects of the interface and communicates with the application in an event-driven fashion, i.e. a listener is called every time an event, such as a mouse click on a button, occurs.

[32] **A**pplication **P**rogramming **I**nterface

For historical reasons, unixiod systems lack such a comfortable and standardised scheme. Most commonly, there is a dedicated process, the so-called X server, that manages only low-level functions of a GUI (e.g. the rendering of a text or the drawing of primitives) and leaves most of the work to the application. For that reason different toolkits were designed to overcome that gap. They show differences in technical and optical aspects of the interface. Nowadays there are at least four common libraries with Perl bindings.

- *Perl-Tk* is a binding for the Tk-toolkit, an old but often available framework. Tk-based GUIs are usually considered unattractive and unintuitive.
- *Perl/Qt* and *gtk2-perl* implement bindings to the two frameworks most commonly used on modern unixiod desktop environments. They offer a comfortable programming interface and produce state-of-the-art GUIs, however, impose significant dependencies to third party binary libraries.
- *wxPerl* is a binding to the wxWidgets library which offers a consistent wrapper for different "native" graphical systems.

None of the frameworks belong to the Perl's core functions and hence require the corresponding Perl packages as well as external libraries. In order to increase the portability of the CTS's GUI, a different approach was chosen: The interface is implemented using omnipresent web technologies, such as HTTP, XHTML, CSS, and JavaScript.

A document defined with the XHTML[33] is typically displayed by a web browser and can include annotated texts, images, controls, such as buttons, tables and other multimedia contents. The page's body is described by an object tree with elements of different types. The textual representation of such an element in the source code is called *tag*. While XHTML is supposed to describe only the semantics of a page, CSS[34] determines the content's presentation.

For instance, it influences the colour, font and size of a text. JavaScript[35] is the only programming language supported by virtually all modern browsers. It can manipulate the document's structure and style, and react to events.

The source code of all three languages is directly interpreted by the browser. All technologies are standardised and hence are supposed to work on each compatible platform. No compiler producing machine depend code is required.

---

[33] **E**xtensible **H**yper**T**ext **M**arkup **L**anguage, uses XML to represent a document based on a tree grammar.

[34] **C**ascading **S**tyle **S**heets

[35] builds on top of ECMAScript, standardised under ISO/IEC 16262:2002. The core functionality of the language is compatible throughout all engines. Typically, there exist incompatible browser-specific extensions to the language, that are not used by the CTS's GUI to achieve a better portability

Since the document and referenced multimedia contents are stored in ordinary files, they are directly accessible from a local web browser. However, a most crucial concept of the World Wide Web is the possibility to request files on a remote machine. These read requests are handled by HTTP[36], a protocol on the seventh layer of the ISO/OSI model (see section 3.2.1). It is a connectionless and file orientated protocol between a client and a server.

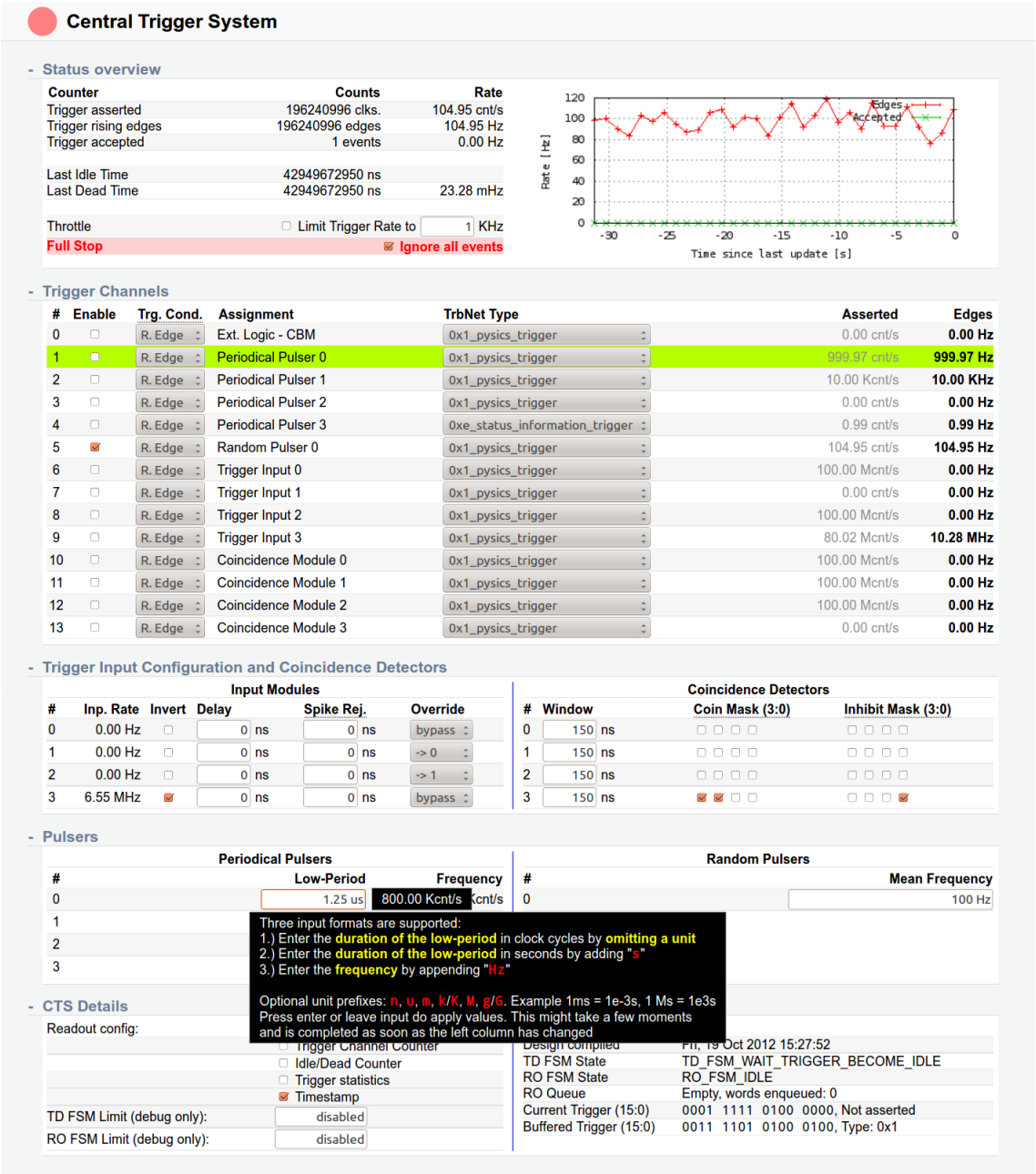

Figure 14: Screenshot of GUI while currently editing the interval of a periodical pulser. Note the highlighting of related elements and the context sensitive information provided to the user.

[36] **H**yper**T**ext **T**ransport **P**rotocol

A connection is always initialised by the client sending a request to a server which – under simplified circumstances – terminates the connection as soon as the complete response is sent. The protocol defines several access methods to retrieve, investigate and alter the server's data. The CTS GUI requires only the simplest and most frequently used one: The `GET` method requests the content of a file identified by a relative path.

Alternatively, the requested file can be a program that is executed by the server each time a client issues a request. The program's output is embedded in the server's response, allowing to dynamically generate documents. This is a concrete implementation of an RPC[37] scheme. Just like in any other ordinary function call concept, it is possible to provide custom parameters to influence the behaviour of the function called.

4.2.1 The GUI's infrastructure

The decision to implement the GUI with web technologies has important implications on its structure:

1. There is one central server that connects to the CTS. It requires Perl 5.12 (or higher), the `HADES::TrbNet` components and uses a gnuplot[5] installation, if available. There is, however, no need for a graphical system. The web server is based on HTTPi[6] which is fully implemented in Perl. No installation or even elevated user privileges are necessary. This is an important advantage over commonly used servers, such as Apache, which is much more versatile but harder to configure.

2. In theory – an arbitrary number of clients can access the server. A client's machine requires only a web browser with JavaScript enabled. Connections from ordinary PC workstations, tablets and smart-phones were tested during development. The server and client process can – of course – be executed on the same machine.

4.2.2 The server component

For technical reasons, the server is split into a number of interacting processes. While this necessity introduces a certain complexity, there is little to learn about the CTS tools from a rather technical, in-depth analysis of the server component. Thus, this work focuses more on the client part and only roughly describes the server's structure:

It consists of an HTTP server process and a monitoring instance of the CTS low level tool chain (see section 4.1.4). As the former process has to distribute the data collected by the monitoring daemon, an interprocess communication (IPC) scheme had to be implement.

[37] **R**emote **P**rocedure **C**all

When initially connecting to the CTS, the daemon stores all information required to repeat the enumeration without actually accessing the device into a dedicated file. It includes the address of the TrbNet daemon, the CTS's endpoint id as well as the headers of the trigger logic.

Furthermore, the monitor process updates a second file containing a complete status snapshot of the device approximatively once per second. If available, the daemon uses a process of `gnuplot`[5] to illustrate the recent developments of the trigger rates.

All files lie within the HTTP server's document directory and are hence accessible from any client connected. The data is stored in the JSON[38] format, which is easy to parse for a web browsers. The images can be produced in the PNG[39] or SVG[40] format. While – in theory – the later format has advantages regarding scaling and font rendering, it produces bigger files and is not yet equally well supported by common web browsers.

In order to manipulate the CTS's settings, a dedicated script on the server accepts RPCs via HTTP to selectively read or write registers or slices. It further delivers a structured representation of the CTS's enumeration results. For security reasons, it is not possible for the client to specify the endpoint's id or the TrbNet daemon's name. This information is recovered from the enumeration cache mentioned above.

There is a central start up script that starts all server processes needed. If supported by the operating system, the script further tries to place the exchange file into a "RAM disc" and, thus, to prevent frequent disk accesses. In Linux, this is implemented via soft links into the *shared memory* file system.

4.2.3 The GUI

The graphical user interface offers all information and settings on a website. If the server is running, a user simply has to navigate to the address in order to "start" the application. A screenshot of interface is shown in figure 14. To minimise the need of navigation through different application screens, the interface consists of a single form that includes all information and parameters. Related values are grouped together. For instance all options of a given trigger module are placed within a dedicated tab. If a component is not required, it can be hidden to reduce the page's size and thus to reduce scrolling.

The page's structure adopts to the device's synthesis settings, e.g. to reflect the number of components. There are two common schemes to implement such a dynamic page structure:

[38] **J**ava**S**cript **O**bject **N**otation, a powerful, compact and human-readable notation for complex data structures

[39] **P**ortable **N**etwork **G**raphic, a lossless compressing bitmap image format

[40] **S**calable **V**ector **G**raphic, a XML-based open vector format

- For a typical internet application, the web server gathers the information required and subsequently produces the HTML code on the fly. This is often implemented using a template engine that allows for a comfortable description of the data-driven parts of a page. The big advantage of this scheme is that there are little requirements for the browser: As it receives the final document structure, no further manipulations of the page are necessary.

  At the same time, this can cause problems if the data presented is subject to regular changes. For each update, the whole document has to be be reloaded. While modern browsers usually restore the view, i.e. scroll to the position that was selected before the reload took place, this scheme renders the usage of textual input elements impossible: If an update is issued while the user types in a value, the input gets lost. Furthermore, a complete reload increases the processor and network usage for the server and the client.

- The concept of AJAX[41] [13] targets those issues. In this scheme, the client receives a static skeleton containing a program to separately request a (possibly abstract) representation of the dynamic data.

  The embedded script then integrates the data into the document tree. Such a request can be issued directly after the page is initially loaded, or as a response to events, such as an user input or a timer.

As the CTS GUI has a number of input fields the first approach is no option. A combination of both schemes seems also inadvisable, because it leads to similar code structures that have to be implemented in multiple languages. Therefore, the server delivers a page template which is altered by the client's browser to fit the CTS configuration. The skeleton already contains static values that are present in each CTS, such as the trigger rate and debugging information.

There are three different RPC types, all of which are handles by a single server-side script:

- The `init` command is issued by the client shortly after the skeleton was received. It returns a structured description of the CTS. Using Perl's introspection capabilities the JSON package is able to traverse the object-orientated representation of the device (see chapter 4.1). The translation unit ignores all object methods and sends only the attributes of each instance to the client. While this generic approach

[41] **A**synchronous **J**avaScript **a**nd **X**ML, describes the idea of requesting new data on demand without reloading the complete page. Initially only XML, which allows tree transformations based on XST, was used as the data exchange format. Nowadays the more compact and flexible JSON is covered by the acronym as well.

may gather information not required by the GUI, the generic method justifies that overhead – after all, the complete response for a typical CTS has a size of less than 30 KB.

- The `read` command can be invoked to read registers and slices for special occasions and further developments. Currently, the GUI is based exclusively on the data recorded by the monitoring process which results in a constant load for the TrbNet independently of the number of active clients.

- The `write` command is issued as soon as the user changes the value of an input element. Each request can contain any number of register and slice values. If a register is not fully defined, the script automatically fetches the register's current state and updates only the slices defined.

  Currently, this can lead to a race condition if multiple requests are issued at the same time. There are, however, no real practical implications, as the read-manipulate-write cycle requires less than 10 ms and the number of write requests issued is very low. It further is unlikely that two operators change the same settings at the same time without causing more severe problems. It, therefore, seems plausible not to implement a locking mechanism for the benefit of a reduced code complexity. [42]

4.2.4 Mapping data

The central task of the client side program is routing the data received to the corresponding positions on the page. In a classical approach, one defines uniquely identifiable elements inside

the document and then implements a static mapping between those artificial identifiers and the slice's name. Such a mapping is often implemented with a dedicated script. The CTS uses a different, more data-driven approach:

In contrast to its predecessors, XHTML allows the definition of custom attributes for document elements. This is used to allow a direct link between any tag and a slice or register value. The `slice` attribute defines data's origin in the format `register.slice[bit]`. The slice component is optional. If omitted, the whole register is used. The bit selection is available only for bitmasks.

Each time new data arrives, an algorithm scans for elements of classes `autoupdate`, `autorate` and `autoratevalue` and replaces their content with the new values received. The tag's class specifies whether an ordinary register, a counter or a rate derived from

[42] Recent changes of Perl's TrbNet bindings now allow for atomic updates of those values which can be used to avoid the race conditions described. This new feature will be in the next major update.

```
<!-- These table cells are automatically updated with the number of
     events accepted. The hard coded content "n/a" will only appear,
     if there is an error while fetching the data -->
<td class="rate  autorate" slice="cts_cnt_trg_accepted.value">n/a</td>
<td class="value autorate autoratevalue"
    slice="cts_cnt_trg_accepted.value"
    suffix=" events">n/a</td>

<!-- Checkbox to enable the 3rd trigger channel. The element was
     generated by the GUI backend and exported in the XML format. A
     click on the checkbox sends the updated status to the web
     server without any further code. Note the title which is
     automatically added to each element with the slice attribute.
-->
<input type="checkbox" class="autocommit autoupdate"
       slice="trg_channel_mask.mask[2]"
       title="trg_channel_mask.mask[2]: Address 0xa101 Bit 2" />

<!-- Example of callback functions. While this example specifies the
     functions within the attribute, it is possible to reference
     external function by their name -->
<input class="autocommit autoupdate" slice="cts_throttle.threshold"
       format="var f=function(x){return parseNum(x)+1}; f"
       interpret="var f=function(x){return parseNum(x)-1;}; f" />
```

Listing 3: Examples of the automated data mapping. The code is taken directly from the GUI and is fully working.

a counter has to be displayed. There are options to define pre- and suffixes and append scaling units. To realise non-standard formats a call-back function can be specified that is invoked before each update.

A second function scans for input elements with the `autocommit` class and registers an event listener that automatically forwards changes to the server. As above, a call-back function may be used to interpret the user inputs. The pulser rate inputs, for instance, make use of that feature to allow for inputs in different units, such as hertz and seconds.

All elements that have a valid `slice` attribute are further assigned a `title` attribute which causes a browser to show a *mouse over hint*. It informs the user about the data source, i.e. the register address and bits used for the slice (listing 3 shows an example). This information can be useful if a register has to be accessed via a third party tool.

5 Summary

This work discussed the design and implementation of a new Central Trigger System for TrbNet based networks with an emphasis on the TRB3 hardware. As this board is intend as a general purpose platform for various experiments, a flexible and extendible trigger system was favoured over a hard to control universal solution. The design allows to operate the CTS as a triggering master or as a triggered slave. In slave mode, the CTS manages a subsystem of a detector that is operated using another DAQ structure. This, for instance, may be required in the case of a TDC upgrade for an existing experiment.

The new trigger system eliminates the need for a dedicated CTS board as it can be embedded into the central FPGA of a TRB3 board that formerly only contained a TrbNet hub. This allows for a complete trigger and read out system on a single board without reducing the sufficiently demonstrated scaling capabilities of the TrbNet.

The structure of the hardware description as well as its VHDL implementation support the extension via modules. A number of functions based on that concept, such as coincidence detection, regular and random pulsers, have been developed. New trigger modules can either be connected directly to the IOs of the chip or can rely on the central input processing which offers features such as spike rejection or run-time compensation.

A bridge to the CBM / MBS network was included and successfully tested at the CBM RICH test beam at CERN in the beginning of November 2012. Furthermore, the new CTS successfully triggered a PANDA test run with 10 TRB3s and $\approx$ 2.500 TDC channels.

A memory structure that resembles the flexible module structure was defined. It allows to remotely identify the capabilities of a given CTS without the need of maintaining a separate configuration database. The data-driven enumeration process executed by a remote software solution can even be completed if the CTS contains unknown or unsupported modules.

The software solution developed builds on top of the existing TrbNet stack and imposes little to no library dependencies. The software defines and employs a generic framework for modelling the register layout of any TrbNet endpoint. This enables the access to the device's properties via meaningful names.

It further offers a read and write cache, as well as a simple type system to perform sanity checks of data exchanged with the remote device. The software allows to define drivers for new hardware modules. In many cases such a driver can rely on a variety of generic functions and hence typically consists of only a few lines of code.

A comfortable and interactive graphical user interface was created using modern web technologies which are accessible via a preconfigured and light-weight web server included in the software stack. This access pattern allows for distributed monitoring and controlling of the CTS from virtually any web-enabled computer or mobile phone.

5.1 Outlook

The new CTS offers basic features that should be sufficient to control simple experimental setups. The need for more complex trigger criteria may arise "in the field" and should be easily satisfiable either using an additional trigger module, or external analogue circuits which cannot be implemented using FPGAs. Currently, bindings for the WASA experiment are planned.

The migration of a TDC module (see chapter 3.1.7) into the CTS can improve its accuracy. It might be useful for master and slave mode operation. In the latter case, it can be used to reduce the jitter between the foreign trigger scheme and the TrbNet subsystem. The CBM/MBS already includes an asynchronous output for that purpose that indicates the arrival of the first data bit.

At this point, the CTS supports only a single event builder which might impose a bottle neck for experiments with data rates exceeding the performance of a single storage server. In order to support such high rate setups, a scheduler can be added to the network logic, e.g. using the simple round-robin approach. This extension can be implemented without breaking the compatibility to the current design.

While the software's foundation design has not undergone severe changes since its modelling, the higher levels of the application logic show symptoms of an organic growth. There are a number of generic functions, such as the monitoring features or the interface functions between the user interface and the foundation, that might be useful for other projects as well. Hence, an encapsulation of such structures into dedicated packages could ease the development of tools for other hardware. This also applies for the generic components of the client side scripts.

As a feedback of the beam test, a faster update of the graphical interface was requested. As discussed in section 4.1.4, the refreshing frequency, however, is a compromise between accuracy, system load and user comfort. The main issue is that after tweaking a CTS setting, in the worst case the user has to wait up to 2 s to see the first results. A decrease of the monitor's update interval, increases the relative error induced by statistical effects.

Thus, an interleaved sampling can be introduced. In the simplest case, one doubles the update interval, but calculates the rates based on the most recent and the and two reads before that. Hence, the latency shrinks while the averaging window remains constant.

Using methods of numeric differentiation, the accuracy can further be increased if the rate is interpolated as a weighted sum of multiple updates.

In order to reduce the network and the client's processor load, it is further possible to send only those values from the server to the client, that changed since the last request. Additionally, a migration of the plot rendering from the server to the client can decrease the network load and lead to a better annotated visualisation. It, however, breaks the compatibility with older browser.

References

[1] *Erklärung gemäß §29 (12) der Bachelor Pfrüfungsordnung – Fassung 2011.* http://www.uni-frankfurt.de/fb/fb13/pruefungsamt/Dateien/Erklaerung_Para30_11.pdf

[2] *Hades Wiki: DaqSlowControl / TDCReadoutBoardV3.* http://hades-wiki.gsi.de/cgi-bin/view/DaqSlowControl/TDCReadoutBoardV3

[3] *ATLAS Fact Sheet.* http://www.atlas.ch/fact-sheets-view.html. Version: 2012

[4] *ATLAS trigger system.* http://www.atlas.ch/trigger.html. Version: 2012

[5] *Gnuplot: homepage.* http://www.gnuplot.info. Version: 2012

[6] *HTTPi: "the little 100% Perl webserver that does tiny things".* http://www.floodgap.com/httpi/. Version: 2012

[7] Frank Lemke, et a.: A Unified DAQ Interconnection Network With Precise Time Synchronization. In: *IEEE Transactions on Nuclear Science* 57 (2010), S. 412–418. http://dx.doi.org/10.1109/TNS.2010.2042176. – DOI 10.1109/TNS.2010.2042176

[8] Friese, Volker (Hrsg.) ; Sturm, Christian (Hrsg.): *CBM Progress Report 2011.* GSI Darmstadt, 2011. – ISBN 978–3–9811298–9–2

[9] Grzegorz Korcyl, et al.: *A Users Guide to the TRB3 and FPGA-TDC Based Platforms.* 2012

[10] IEEE: *IEEE Standards Interpretations: IEEE Std. 1076-2002, IEEE Standard VHDL Language Refernce Manual*

[11] J. Michel, et al.: The upgraded HADES trigger and data acquisition system. In: *Journal of Instrumentation* 6 (2011)

[12] J. Michel, et al.: *A Users Guide to the HADES DAQ System.* 2012

[13] Jesse James Garrett: *Ajax: A New Approach to Web Applications.* http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. Version: 2005

[14] Lattice Semiconductor Corp.: *LatticeECP2/M Family Data Sheet – DS1006 Version 03.9, January 2012*, www.latticesemi.com/documents/DS1006.pdf

[15] Lattice Semiconductor Corp.: *LatticeECP3 Family Data Sheet – DS1021 Version 02.2EA, April 2012*, www.latticesemi.com/documents/ds1021ea.pdf

[16] M. Traxler et al: *A compact system for high precision time measurements (<14 ps RMS) and integrated data acquisition for a large number of channels.* 2011

[17] Michel, J.: *Development of a Realtime Network Protocol for HADES and FAIR Experiments*, Uni. Frankfurt, Institut für Kernphysik, Diplomarbeit, 2008

[18] Michel, J.: *Development and Implementation of a New Trigger and Data Acquisition System for the HADES Detector*, Uni. Frankfurt, Institut für Kernphysik, Diss., 2012

[19] Sauli, F.: *Principles of Operation of Multiwire Proportional and Drift Chambers: Lectures Given in the Academic Training Programme of CERN.* European Organization for Nuclear Research, 1977 (CERN (Series))

[20] Schadmand, S.: Wasa at Cosy. In: *ArXiv Physics e-prints* (2005), November

[21] The HADES Collaboration (G. Agakishiev et al.): The high-acceptance dielectron spectrometer HADES. In: *THE EUROPEAN PHYSICAL JOURNAL A* (2009)

[22] Zimmermann, H.: OSI Reference Modell - The ISO Model of Architecture for Open Systems Interconnection. In: *IEEE Transactions on communication* 28 (1980), April, S. 425–432

A Appendix

A.1 Slow Control Registers

Address	Bit(s)	Description
0xa000		Statistics: Number of clock cycles with trigger asserted
0xa001		Statistics: Number of trigger rising edges
0xa002		Statistics: Number of triggers accepted
0xa003		Current trigger status
	15: 0	Trigger bitmask (before filtering)
	19:16	Current trigger type
	20	Trigger asserted
0xa004		Buffered trigger status
	15: 0	Trigger bitmask (before filtering)
	19:16	Trigger type
0xa005		TD FSM State (Trigger Distribution). One-Hot-Encoding:
	0	TD_FSM_IDLE
	1	TD_FSM_SEND_TRIGGER
	2	TD_FSM_WAIT_FEE_RECV_TRIGGER
	3	TD_FSM_FEE_ENQUEUE_INPUT_COUNTER
	...	
	12	TD_FSM_WAIT_TRIGGER_BECOME_IDLE
	13	TD_FSM_DEBUG_LIMIT_REACHED
0xa006		RO FSM State (Readout Handling). One-Hot-Encoding:
	0	RO_FSM_IDLE
	1	RO_FSM_SEND_REQUEST
	2	RO_FSM_WAIT_BECOME_BUSY
	3	RO_FSM_WAIT_BECOME_IDLE
	4	RO_FSM_DEBUG_LIMIT_REACHED
0xa007		Readout Queue
	15: 0	Words enqueued
	30	Empty
	31	Full
0xa008		Debug FSM limits
	15: 0	Number of Triggers (0xFFFF means no limit)
	31:16	Number of Read-Outs (0xFFFF means no limit)
0xa009		Trigger information to be send in read-out (default: 0x00000000)
	0	Input Counters
	1	Channel Counters
	2	Statistics: Idle- and Dead-Time counter
	3	Statistics: Trigger asserted, -edges, -accepted
	4	Timestamp
0xa00a		Statistics: Dead time of last trigger
0xa00b		Statistics: Time between last two accepted triggers
0xa00c		Event throttle
	9: 0	Maximal number of events accepted per millisecond
	10	Throttle enabled
	31	Stop Trigger

Table 4: Registers with fixed addresses

A.2 CTS Tool - Plug-In File

```
1  package CtsMod30;
2  @ISA = (CtsBaseModule);
3  use warnings, strict, TrbRegister;
4
5  sub moduleName {"Periodical Counter"}
6
7  sub init {
8     my $self    = shift;
9     my $address = shift;
10    my $cts  = $self->{'_cts'}
11    my $regs = $self->{'_registers'};
12    my $prop = $self->{'_properties'};
13    my $header = $cts->{'_enum'}{0x30}->read();
14 # END OF TEMPLATE
15
16 # registers
17    for(my $i = 0; $i < $header->{'len'}; $i++) {
18       $regs->{"trg_pulser_config$i"} = new TrbRegister($address+1+$i,
19                                                        $cts->getTrb, {
20          'low_duration' => {'lower' =>  0, 'len' => 32},
21       }, {
22          'label' => "Periodical Counter Configuration $i",
23          'accessmode' => "rw",       # control register:  read/write access
24          'monitor' => 1,             # regularly fetch in monitoring mode
25          'export' => 1               # save value when exporting CTS config
26       });
27    }
28
29 # human-readable itc assignment
30    for(my $i = 0; $i < $header->{'itc_len'}; $i++) {
31       $cts->getProperties->{'itc_assignments'}[$i + $header->{'itc_base'}]
32          = "Periodical Pulser $i";
33    }
34
35 # properties
36    $prop->{"trg_pulser_count"} = $header->{'len'};
37    $prop->{"trg_pulser_itc_base"} = $header->{'itc_base'};
38 } 1;
```

Listing 4: This listing shows the complete plug-in file loaded if a Periodical Counter Trigger Module (Header ID: 0x30) is present. Each plug-in has to implement the `moduleName` method, which is used for introspection purposes, and the `init` method, that defines all registers and properties. Line 1 to 16 are identical (with expection to the header id and the module's name) in all plug-ins. Registers and properties are created as attributes of the plug-in instance. They are automatically linked to the central file by the parent class `CtsBaseModule`. This might, again, be useful for introspection purposes, as it allows to determine which registers are defined by a plug-in during run-time.