Improving software efficiency by changing the FELIX data format

Jochem Leijenhorst (500855372)

June 25, 2024

Supervisor Nikhef
Frans Schreuder

Supervisor Hogeschool van Amsterdam
Caspar Treijtel

Hogeschool van Amsterdam

Summary

Connected to CERN's Large Hadron Collider (LHC) is a particle detector known as ATLAS. This particle detector produces a large amount of data, which needs to be collected. To do so, a data acquisition system called FELIX is used. FELIX collects data from the particle detection electronics by using an FPGA on a custom PCIe card. The card transfers this data to a server which publishes it onto a network. The data transfer from card to server has a certain format. This format was shown to be suboptimal; changing it could improve the efficiency of the software running on the server.

Three different new formats were proposed, two of which were simulated. The simulated formats improved the efficiency significantly. Because the more efficient of the two new formats was an extension of the other one, the latter was implemented first.

The format specifies chunks of data starting with a header, which contains the length of said chunk of data. The new format was implemented on both the software and the firmware. The firmware utilizes a FIFO to store the chunks of data, and does not yet know the length of one at the start of the chunk, making it impossible to produce this format with a conventional FIFO. This required a special solution: a FIFO in which a space can be reserved for a header, the contents of which are inserted after the entire chunk of data has been inserted.

After the implementation was made, it was tested. It produced the right data format, and the server could parse it correctly. For one firmware flavour [1] it showed a significant improvement in efficiency. Using another firmware flavour showed no improvements, however. The reason for this will need to be researched.

Samenvatting

Aan CERN's Large Hadron Collider (LHC) zit een deeltjesdetector genaamd ATLAS verbonden. Deze detector produceert een grote hoeveelheid data, die allemaal verzameld moet worden. Dit doet ATLAS door middel van een data-acquisitiesysteem genaamd FELIX. FELIX verzamelt data van de detector door middel van een speciaal gemaakte PCIe kaart met daarop een FPGA. Deze kaart stuurt de data vervolgens op naar een server, die de data vervolgens naar een netwerk publiceert. De dataovergang tussen de kaart en de server gebruikt een bepaald dataformaat. Volgens onderzoek is dit formaat suboptimaal. Met een beter formaat zou een hogere efficiëntie bereikt kunnen worden.

Er zijn drie verschillende formaten voorgesteld. Twee hiervan zijn gesimuleerd, en blijken efficiënter te zijn dan het originele formaat. Van de twee formaten is de efficiëntere een aanpassing van de minder efficiënte. Hierdoor is eerst het minder efficiënte formaat geïmplementeerd.

Het te implementeren formaat specificeert data chunks die beginnen met een header die de lengte van de data chunk bevat. Dit nieuwe formaat is zowel in de software als de firmware geïmplementeerd. De firmware gebruikt een FIFO om de data chunks in op te slaan, en weet de lengte van de chunk nog niet wanneer het eerste gedeelte hiervan de FIFO in gaat. Dit maat het onmogelijk om met een standaard FIFO het nodige formaat te produceren. Hier is een oplossing voor bedacht: een speciale FIFO die een plek in zijn geheugen kan reserveren voor een header, zodat deze later geschreven kan worden wanneer

[1] A firmware flavour is a certain set of settings for the firmware on the FPGA

de volledige chunk al in de FIFO zit.

Na de implementatie gemaakt te hebben is deze getest. Er kwam een datastroom uit met het correcte formaat. Voor één van de firmware flavours[2] is de efficiëntie van de software significant verbeterd ten opzichte van de efficiëntie van het originele formaat. Met een andere firmware flavour is de efficiëntie echter niet verbeterd. De reden hiervoor zal nog onderzocht moeten worden.

[2]Een firmware flavour is een bepaalde instelling voor de FPGA firmware

Contents

1 Introduction

Nikhef is a research institute based in the Netherlands that focuses on research into subatomic physics. Apart from physicists Nikhef also employs engineers to work on various experiments related to particle physics. Among these experiments are the particle detectors at CERN. These are made to be able to track the particles generated when a particle collision occurs. The ATLAS experiment, shown in fig. 1, is one of these detectors.

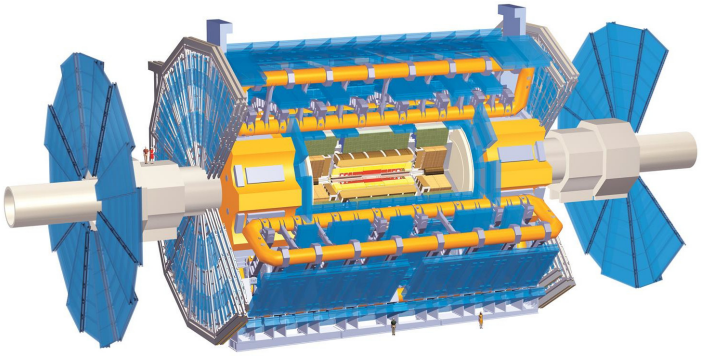


Figure 1: A computer generated image of the ATLAS detector [1].

ATLAS produces an incredibly large amount of data, from multiple different types of particle detectors. The data these detectors produce needs to be transferred to a computer network outside of ATLAS in order to be stored. The detectors output their collected data through optical links, using multiple different formats. To be able to transfer the data over one network, it needs to be read, decoded, and finally published onto the network. Within ATLAS, this part is handled by the Front-End LInk eXchange (FELIX).

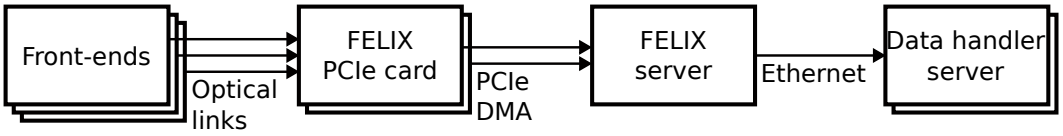


Figure 2: A small diagram of the FELIX interface.

FELIX collects data from the detector electronics (also referred to as front-ends), decodes the data to raw bytes, and then sends it through a commodity high bandwidth network [2]. A block diagram of FELIX is shown in fig. 2. FELIX uses a custom-designed PCIe card, shown in fig. 3, featuring a Xilinx Ultrascale FPGA. This card collects the data from the front-ends and sends it to the FELIX server (shown in fig. 2) through DMA. This server then publishes the data it received to the network.

A block diagram of the firmware of the FELIX PCIe card can be seen in fig. 4. The incoming data is provided by the front-ends through optical links. Each optical link can contain one or more E-Links. Each E-Link produces a stream of data 'chunks'. These data chunks go through a decoder, which converts every supported data format into AXI streams[3]. The data from these streams is placed into data 'blocks' of a fixed size by the To Host Central Router (CRToHost). CRToHost sends these blocks to Wupper, which directly injects them into the memory of the FELIX server using DMA. The FELIX server then parses these data blocks using the FELIX software.

[3]AXI streams are streams of data, used to communicate between different parts of FPGA firmware.

Figure 3: The FELIX PCIe card, FLX712 [3].

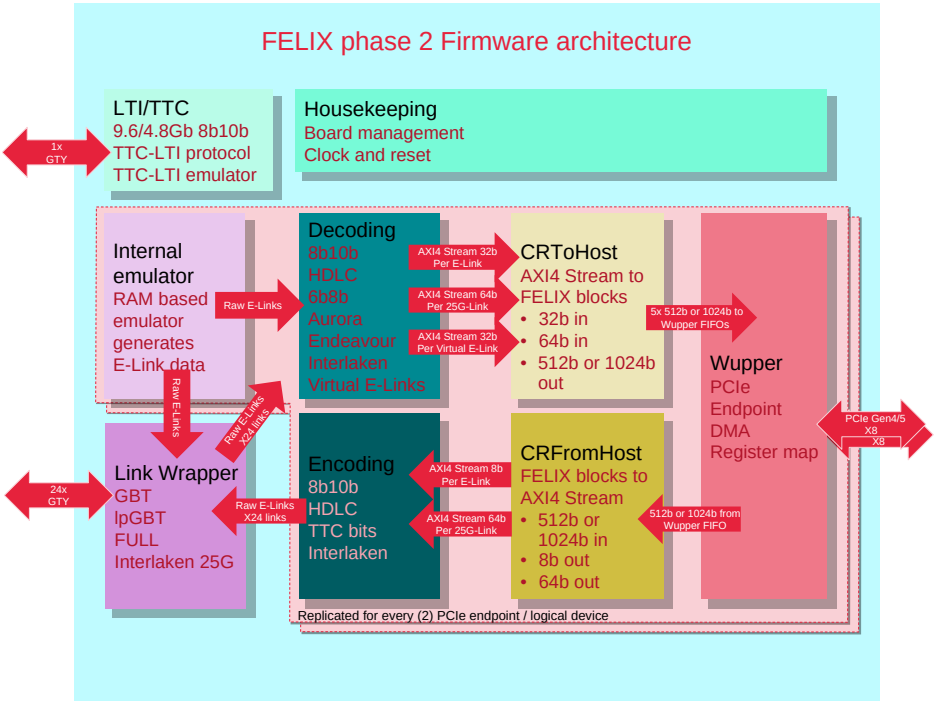


Figure 4: A diagram of the FELIX firmware.

The data blocks CRToHost outputs are of particular note. It has been shown that the speed at which the software can parse these data blocks is limited by the way they are formatted [4]. Because of this, a suggestion was made to change this format, which could allow for faster data transfer rates. This is the topic of this bachelor's thesis, which will be further described in the assignment description, which can be found in section 2.

After the assignment description, various subjects will be researched in section 3. Among these subjects are the current workings of FELIX, the research that has already been done, and the different block formats that were proposed. Following the research, the specifications of the design that needs to be made will be described in section 4. Next, an implementation for both the software and firmware will be designed and implemented in section 5. In section 6, the results of the the tests that were done will be described and discussed. Finally, section 7 presents a conclusion before recommendations for future work are discussed in section 8.

2 Assignment description

The goal of this thesis is to change the CRToHost block format so that it can be parsed more efficiently. Currently, the CRToHost block format, also shown in fig. 5, is defined as follows: The size of every block is equal, and a multiple of 1 KiB. Every block starts with a block header. Following the block header is raw chunk data. Every chunk ends with a 32 bit chunk trailer, which contains the length of the chunk, among other information.

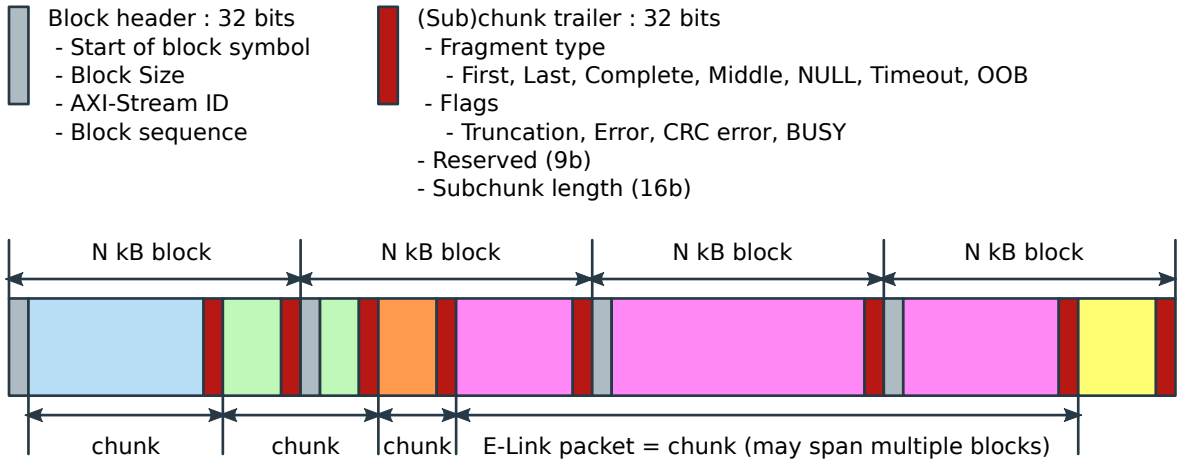


Figure 5: Five chunks spread over four blocks using the ToHost block format [5, p. B.39].

CRToHost uses this format to send the incoming data to the FELIX server. Once received, the FELIX software will read the blocks, collect the chunks within them, and publish these onto the network. To do this, the software needs to pass over each block twice. It first has to find the starting point of every (sub)chunk in a data block by reading every (sub)chunk trailer. When a complete chunk has been collected, it will be published onto the network.

It has been suggested that the software could be made more efficient by only passing over the data once [4]. However, this would require a different block format; with the current format, the software cannot determine the length of a chunk before reading its trailer. Research is necessary to find a better block format. When the new block format presents a good enough improvement, it will have to be implemented to both the FELIX software and firmware.

2.1 Research questions

- Can the efficiency of the FELIX software be improved by changing the ToHost data block format, and if so, how can this be implemented?

This raises the following sub-questions:

1. What is the best way of changing the ToHost block format to improve the efficiency of the software?
2. How much will this improve the efficiency of the software?
3. How can this improvement be implemented into the firmware?

3 Research

This section will discuss the various topics that were researched before any designs were made. Firstly, the current system must be understood before any changes can be made. After this, the numerous proposed formats and the simulations that were done will be discussed.

3.1 Felix

Felix consists of two main parts. The first part is the Felix card, which houses an FPGA. This card takes data from one or more optical links and converts it into data blocks. It then transfers these blocks to the server using Direct Memory Access (DMA) over PCIe. A diagram of the system can be found in fig. 6.

The second part is the FELIX software. This software reads the data the card has placed into the memory buffer, and publishes it to the network.

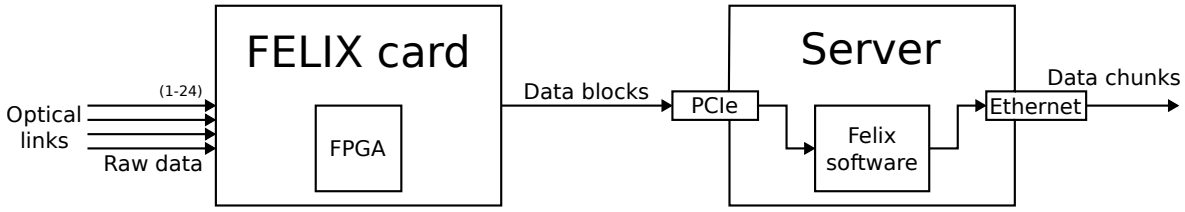


Figure 6: An overview of Felix.

3.1.1 Firmware

The FELIX firmware is responsible for reading data from multiple optical links. Each optical link can contain data from one or more 'E-Links', which can send data using various different protocols and speeds. The firmware consists of multiple different firmware blocks which were designed to handle the incoming data.

The FELIX firmware is kept as generic as possible. This is done so that it can be used for many different purposes. The firmware can be adjusted to work in a slightly different way by changing its 'firmware flavour' [5]. For the purposes of this thesis, only the GBT and FULL mode flavours are of interest. The other flavours are either variants of the two aforementioned flavours or beyond the scope of this thesis.

The GBT mode flavour is available as both an 8 channel and a 24 channel variant. Its purpose is to be able to communicate with a VersatileLink GigaBitTransceiver (GBT) architecture. This architecture was developed as part of CERN's Radiation Hard Optical Link project. It is designed to provide multiple lower bandwidth ELinks from the front-ends through one radiation-hard high-bandwidth data link, running at up to $5\,\mathrm{Gbit\,s^{-1}}$ [2, p. 4].

The FULL mode flavour was made to be able to reach a higher bandwidth data link from the detector to FELIX than GBT can achieve. The FULL mode data link does not require radiation hardness [2]. With FULL mode, each optical link only provides a single ELink.

The Decoding block of the firmware converts the data into data fragments, also referred to as 'chunks'. These chunks are placed into data 'blocks' by the CRToHost block. Each data block only contains chunks from a single E-Link. These data blocks are then

transmitted to a server by the Wupper[4] block over a PCIe connection, by writing it directly into a contiguous memory buffer through DMA.

3.1.2 Software

The FELIX data acquisition software consists of two programs: FELIX-ToHost and SW ROD. A block diagram of the software is pictured in fig. 7.

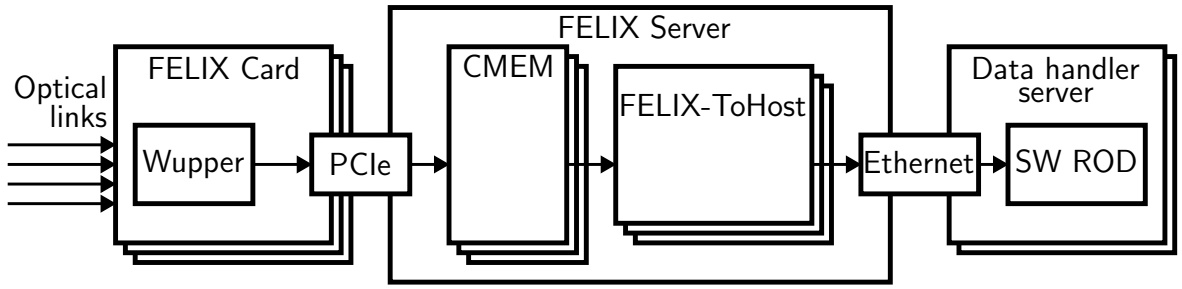


Figure 7: An overview of the FELIX software.

FELIX-ToHost runs on the FELIX server, and is responsible for reading data from the contiguous memory buffer (CMEM). It creates this buffer by using the CMEM library and Linux module, which were created specifically for the allocation of such a buffer [6]. The FELIX card writes this data into the CMEM buffer in blocks, which FELIX-ToHost decodes. Chunks can be separated into subchunks, which can reside in different parts of the CMEM buffer. The chunks are communicated to the data handler server by sending it an array of addresses-size pairs, as shown in fig. 8.

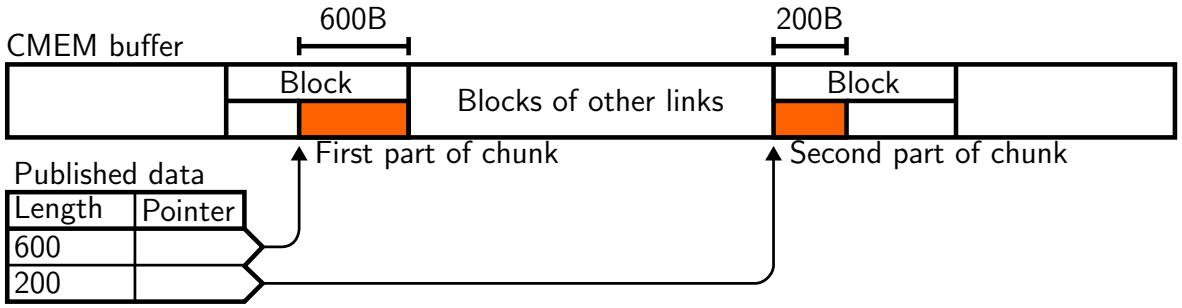


Figure 8: How the FELIX ToHost program publishes data.

SW ROD is a program that runs on a data handler server. It uses the address-size pairs it receives to directly read the chunks from the CMEM buffer by using RDMA network technology [2]. By doing it this way, the FELIX Server never needs to copy data from the CMEM buffer, as the data handler can read it directly from there.

The FELIX toHost program needs to process a very large amount of data blocks in a very short amount of time. Improving it would mean improving the possible data rate FELIX can achieve. It spends a significant amount of time decoding data blocks, which could possibly be reduced by changing the data format these blocks use.

[4]The name 'Wupper' comes from the Groninger sport 'bongelwuppen' [5].

3.1.3 Block format

The CRToHost block within the FELIX firmware (shown in fig. 4) places the chunks of data into blocks of a fixed length. This section will provide a more detailed description of the block format, which was briefly introduced in section 2.

The block format is shown visually in fig. 5. The size of every block is equal, and a multiple of 1 KiB. Every block starts with a block header of 32 bits. The block header is followed by raw chunk data. If the length of the chunk is not a multiple of 32 bits, it is padded accordingly. Every chunk ends with a 32 bit chunk trailer, which contains the length of the chunk, among other information. When a chunk is too large to fit into a block, either by being larger than one or because the block is already occupied by other chunks, it will be broken up into two or more subchunks. This is shown in fig. 5 for chunks B and D.

3.1.4 ToHost

The FELIX-ToHost program needs to decode the format described in section 3.1.3. It does this by iterating over every block it finds in the CMEM buffer, while completing these steps for every block:

1. Go to the end of the block.
2. Find and store the index of the start of every chunk[5] by reading their chunk trailers, thus iterating over the block in reverse.
3. Publish the chunks in order by reading the stored indices.
4. Go to the next block.

Step 2 is only necessary because the program has no way of knowing how long the first chunk is without first iterating over every chunk in reverse. If the chunk trailer could instead be placed at the beginning of a chunk, the program could skip this step entirely. Research was done on this subject, to find out if the efficiency improvement is significant enough to justify implementing the change in both firmware and software. The next paragraph will discuss the results of said research.

3.2 A different block format

A study was done to estimate the improvements offered by one-pass decoding compared to two-pass decoding. In order to answer the first sub-question in section 2.1, this study needs to be discussed. The results of the study were shown in a presentation by Serguei Kolos, the slides of which can be found in [4].

For this study, Kolos simulated incoming PCIe data by copying a pre-made stream of data into the DMA buffer. To test the difference in speed, the block format in the simulation data was changed to have a chunk header instead of a chunk trailer. The software was changed to a single pass algorithm to be able to use this new block format.

Kolos used two methods of testing. The first consists of running the software at 100% CPU, measuring the data speed (fig. 9). The second method consists of measuring the

[5] A block can contain both chunks and subchunks. However, for readability, these will both be referred to as 'chunks'.

CPU usage while varying the number of ELinks, effectively changing the amount of data the program has to process (fig. 10). The results, which can be seen in figs. 9 and 10, showed that a one-pass decoder could achieve a 20 % increase in data rate compared to the two-pass decoder. It uses on average 8.1 % less CPU at the same number of ELinks. This is a significant enough improvement to the performance to justify implementing the change.

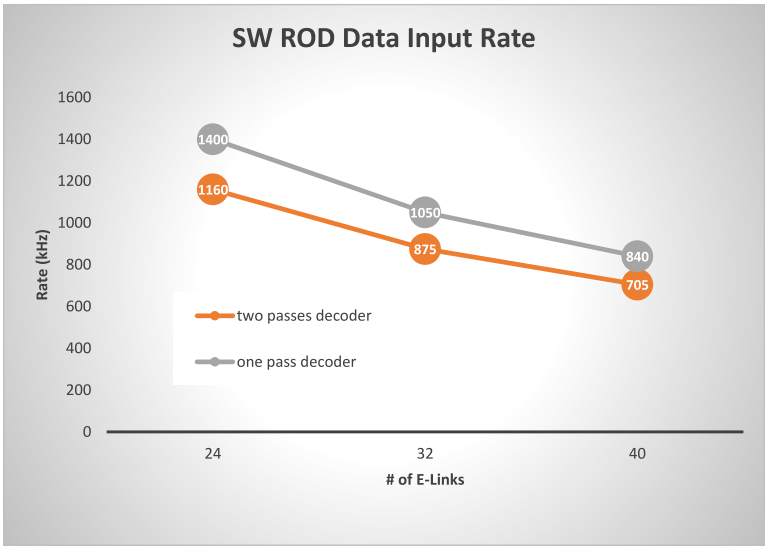


Figure 9: A graph from [4] showing the maximum data rate the FELIX software could achieve by using 100% of a CPU core.

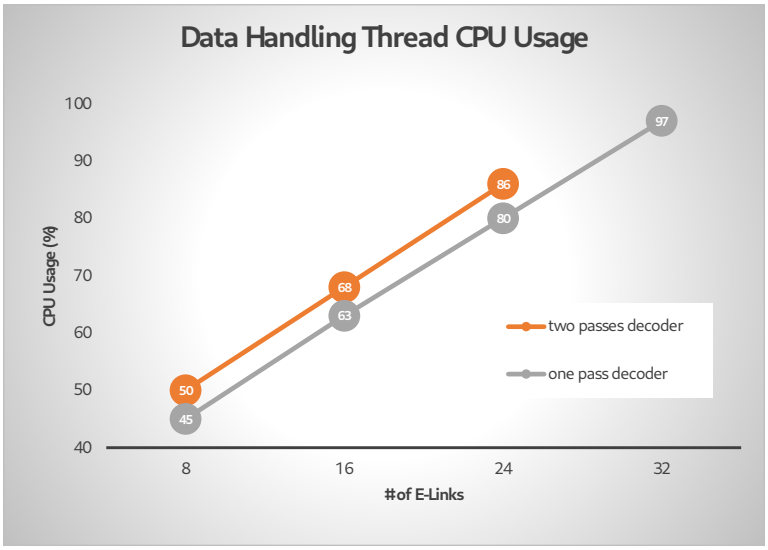


Figure 10: A graph from [4] showing the CPU usage of the FELIX software with differing numbers of ELinks.

3.3 A blockless format

Another approach to increasing the software efficiency is to completely change how data is formatted. If chunks have headers instead of trailers, the parsing algorithm no longer

requires the fixed length of the blocks to find the first header. This removes the need for data blocks, making it possible to only send bare, consecutive chunks which do not need to have a certain fixed length. To prevent a single E-Link from hoarding the output, larger chunks would still need to be split into subchunks when they exceed a certain length.

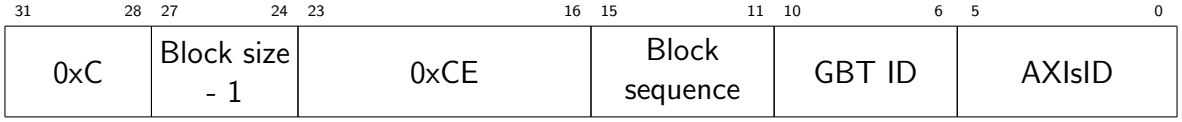
Implementing a blockless format would not necessarily be more complex than implementing a format with chunk headers. It would, however, require a larger amount of adjustments to be made to the current firmware. The firmware currently relies on the fact that data is divided into blocks of equal length.

Using a blockless format could significantly improve the efficiency of the software. Data blocks need to be padded to the right length in certain circumstances, which would not be necessary when only chunks are sent. Furthermore, smaller chunks would never need to be split into subchunks, as they cannot collide with the end of a block.

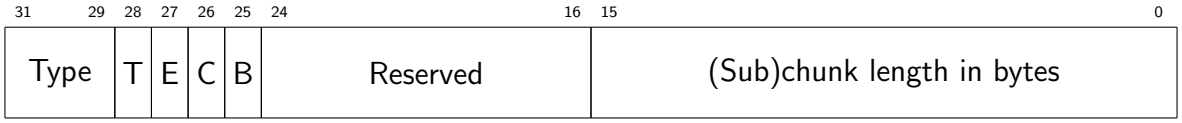
This new format can be tested using the same approach as the test done by Kolos, described in section 3.2. The program Kolos made can be changed to decode a blockless format. This way, the efficiency of the blockless format can be tested without implementing it into the firmware, which will take significantly longer to do.

3.3.1 A new chunk header

To test the format, it first has to be completely defined. It needs to convey the same data the original format did, in a more efficient manner. The original format used both chunk trailers and block headers to convey important metadata about the chunks. Because blocks are no longer used in this case, the vital information within the block headers will need to be placed within the chunk header, together with the information from the original chunk trailer. To do this, the essential fields from both the block header and chunk trailer must be fit into a single chunk header, preferably with a size of 32 bits.



(a) The block header format.



(b) The chunk trailer format.

Figure 11: The original block header and chunk trailer formats.

The original block header format is shown in fig. 11a. The 11 least significant bits in this header, the AXIs ID and GBT ID, are used to identify the E-Link from which the data in the block originates. The next field, called block sequence, is incremented every time a block is sent with data from that particular E-Link. The software uses this to verify that no blocks are lost. The only other field of note is the block size field. This 4 bit field conveys the size of the block in KiB. Because the blockless data format does not contain any blocks, this field will not be used. The rest of the bits in the block header are constants. These are used to be able to easily identify a header in the data, and thus do not carry any vital information.

The chunk trailer, shown in fig. 11b, conveys the length of the chunk in the least significant 16 bits. It also has 4 flags containing important information about the chunk. The three most significant bits contain the type of the chunk. The type of a chunk determines whether it is a subchunk or not, and if it is, the type of subchunk (first, middle or final).

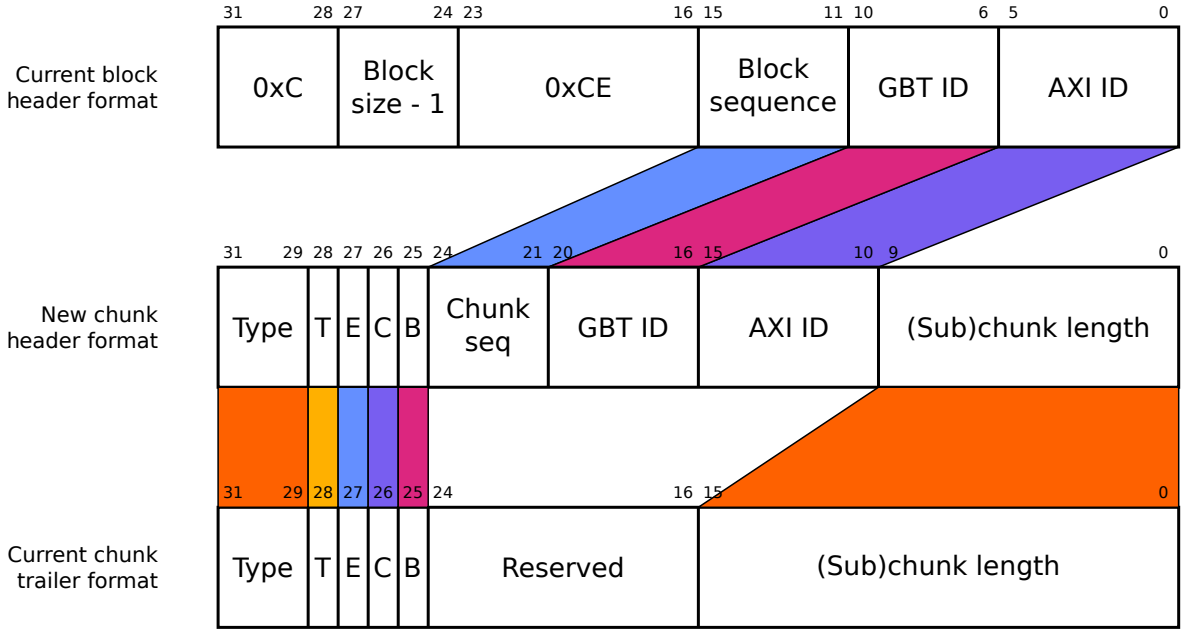


Figure 12: The new chunk header format.

Combining every important field in both the block header and chunk trailer results in the chunk header format shown in fig. 12. Some fields were adjusted to be able to fit into the header. The (sub)chunk length was changed to be 10 bits wide, instead of 16. With 10 bits, a maximum chunk size of 1 KiB can be achieved. If longer chunks are required, they can be broken up into two or more subchunks. Furthermore, the block sequence field was changed to a 4 bits wide chunk sequence field.

3.3.2 Testing the new format

To test impact the different data formats have on the performance of the software, a benchmarking program was used. This benchmarking program was originally written by Serguei Kolos, and works by copying data into the CMEM buffer directly from a file. This way, a certain format can be tested without the need to write firmware that can generate said format.

The benchmarking program was modified to accommodate for a blockless data stream, which required a large amount of changes. The original program relied on the fact that all data blocks are exactly 1 KiB. With the size of the CMEM buffer being a multiple of this, the data blocks can never be truncated. This is not the case when using separate chunks. Without the constant width of the blocks in which they reside, it is possible for the chunks to get truncated by the end of the CMEM buffer, as shown in fig. 13. Such a truncation can only occur once every time the CMEM buffer is filled. Because of the size of the CMEM buffer (8 GiB), any proposed solution to this will take an insignificant amount of time. This means that the test program does not need to take this truncation into account. To test

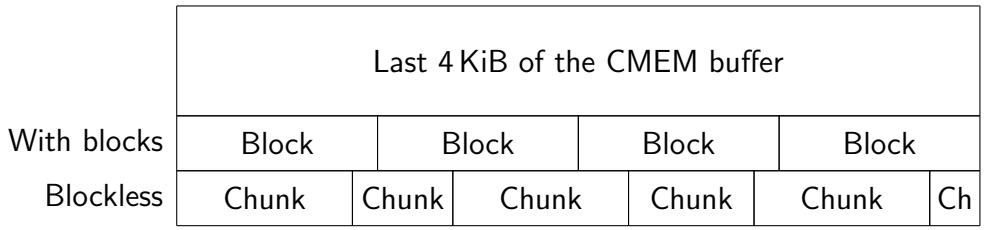


Figure 13: How blocks fit into the CMEM buffer compared to separate chunks.

the new data format without truncations causing problems, the testing data within the data file can be made to fit an integer amount of times into the CMEM buffer. Because this requires a change to the input data, this solution only works for testing. If and when the new format is implemented, a real solution will be necessary.

This data file, along with the data files for the other two formats, need to be created. The data file for the chunk trailer format can be directly taken from a FELIX card, as this is the format currently used by FELIX. To create a data file with the normal chunk header format, a converter program can be used. This program takes every chunk trailer in the file, and moves it to the start of the (sub)chunk. Converting the file to use a blockless format is a much more complicated task. An easier way to obtain a file that uses the blockless format is to generate one. As long as the generated file has the same amount of E-Links, and its chunks have the same lengths as the ones in the other two files, such a generated file would be interchangeable with a converted file.

3.3.3 Results

Two tests were done: one with 100 byte chunks and one with 2000 byte chunks. The results of the tests can be found in figs. 14 and 15. The average proportional improvement compared to the trailer format can be found in table 1. With chunks of size 100 the chunk header format does not appear to improve the efficiency much at all, while with chunks of size 200 the improvement is much greater than the one found in the study done by Kolos [4]. The reason for this is most likely that Kolos either used a different constant chunk size or a varying one.

Chunk size	Headers	Blockless
100	2.8 %	13.5 %
2000	38.2 %	37.9 %

Table 1: The average speed improvement the new formats have compared to the chunk trailer format.

Table 1 and fig. 14 show that the blockless format is much more efficient than the chunk header format when handling shorter chunks. When larger chunks are introduced however, it loses this advantage. This is likely due to the fact that the larger chunks in the blockless format need to be split into subchunks as much as those in the chunk header format. With shorter chunks, only those placed within blocks need to be split into subchunks whenever they collide with the end of the block. The decoding program takes a significant amount of time piecing these subchunks back together.

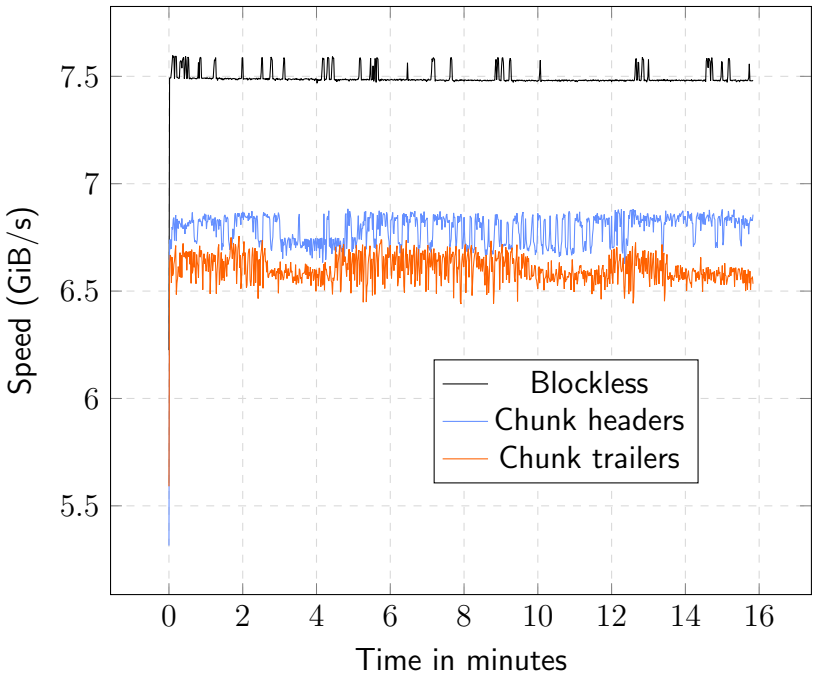


Figure 14: The results of testing all three data formats with a constant chunk length of 100 bytes.

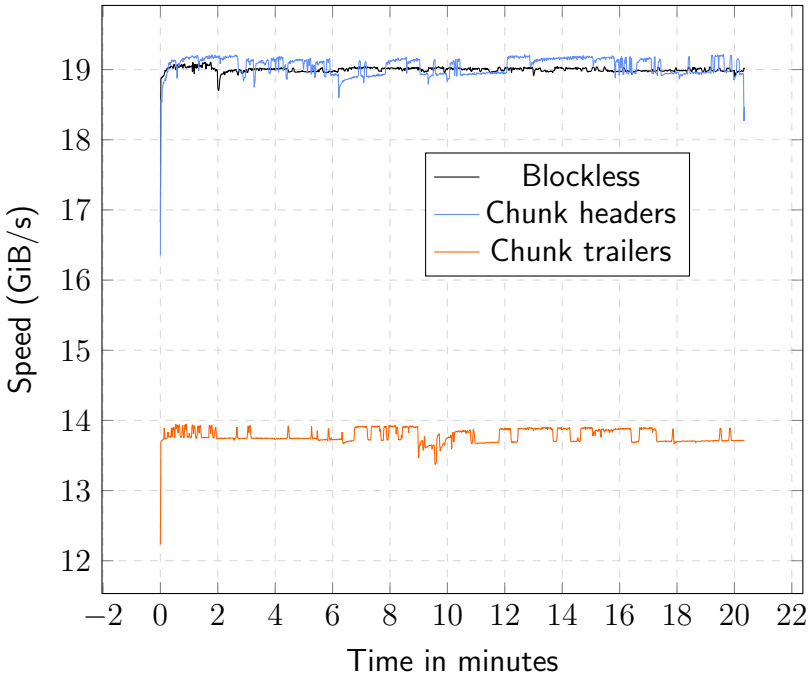


Figure 15: The results of testing all three data formats with a constant chunk length of 2000 bytes.

3.4 A mix of both formats

Another possibility is a mix of the original and blockless formats. Section 3.3.1 showed that it is possible to combine all important information contained within both the block header and chunk trailer within a single 32 bit chunk header. This means that a format that still

uses blocks, but combines chunks of different E-Links into the same block, is possible. This would practically eliminate the use of the padding chunks described in section 3.1.3, which is beneficial to the efficiency of the data transfer.

The advantage of using this instead of a blockless format is that the data is still divided into blocks. The sofware side of FELIX currently heavily relies on this fact, and would need drastic adjustments to account for a blockless format. A blockless format also introduces problems to the parsing process within the software. With a format that uses blocks, whenever the data within one is parsed or written incorrectly, the software can always just move on to the next block. With a blockless format, the software does not have this luxury. Missing one chunk header renders the entire stream of data from that point on unparsable.

Research will need to be done on this hybrid format, to test if it is significantly more efficient than the original format with chunk headers. This research is beyond the scope of this thesis, however, and will need to be done at another time.

4 Specifications

After finishing the research, an implementation must be designed for both the firmware and software. These implementations should be able to generate and parse the new format, respectively. There are a number of specifications these designs need to uphold, which are listed here:

1. The firmware design must be capable of producing the block format as shown in fig. 16. This block format is equal to the original format with its chunk trailers instead being placed at the front of the chunks.
2. The firmware design cannot use more resources than is used by the original firmware.
3. The firmware design cannot use any more memory than is used by the original firmware design.
4. The software design must be able to parse the new format.
5. The efficiency of the software must increase by at least 20%.

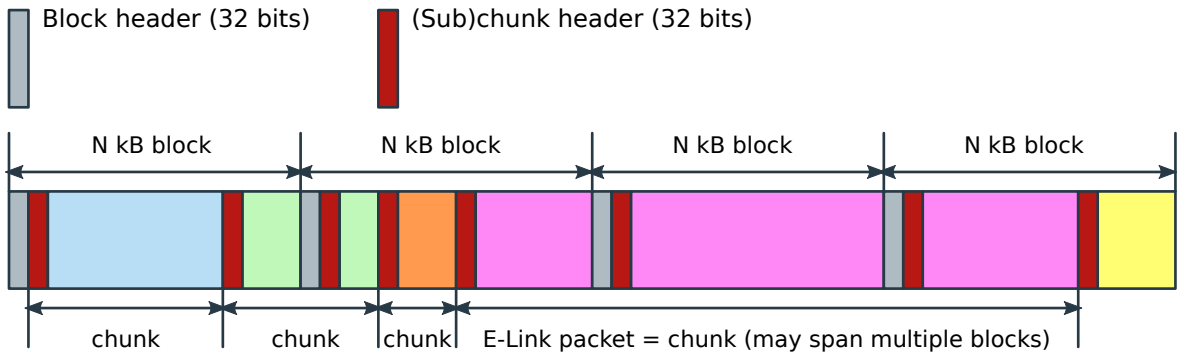


Figure 16: The new toHost block format.

5 Design and implementation

The blockless format described in section 3.3 relies on (sub-)chunks having chunk headers rather than chunk trailers. This allows for the design and implementation of the chunk header generation to be made prior to implementing any of the aforementioned alternative formats.

There are two designs that need to be made for the chunk header generation. First, the CRToHost firmware must be altered so that it can output chunks with headers instead of trailers. The CRToHost firmware is written for an FPGA in VHDL, so the designs described in this chapter will all be implemented in this language. Next, the FELIX software must be altered to handle chunks with headers. After this, if there is enough time, the alternative formats can be further investigated and implemented if they are deemed beneficial enough.

5.1 Header insertion

Inserting a header into the data output is not a trivial task. In the current implementation, chunk trailers are added into the data by the toBlock process within the ToHostAxiStream-Controller, as seen in fig. 17. The data stream is pushed into a FIFO which resides in the CRToHostdm wrapper block.

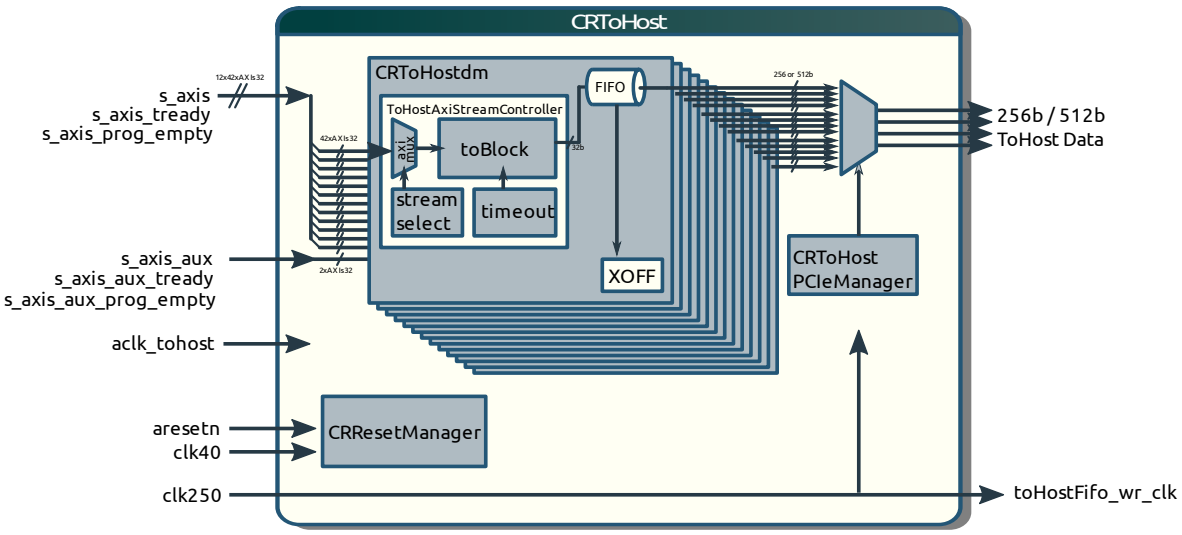


Figure 17: A diagram of the CRToHost block [5].

The toBlock process keeps track of the length of a chunk by counting the amount of data it has processed. The length of the chunk is then placed in the chunk trailer. The toBlock process has no way of knowing the length of a chunk it is going to process before creating this trailer. Because of this, the contents of a chunk header cannot be known at the time it needs to be pushed into the FIFO.

There are multiple possible solutions to this problem. Which will be explained in detail in section 5.1.1.

5.1.1 Intercepting the output

To insert headers into the output of CRToHost, one possible solution is to alternate the output between the FIFO and a register within the ToHostAxiStreamController. This reg-

ister would hold the header for the chunk currently present in the FIFO. As pictured in fig. 18, a mux is required to switch between the FIFO and the header register. This mux would need to be switched by a process that either keeps track of the data or somehow obtains the header position from the ToHostAxiStreamController.

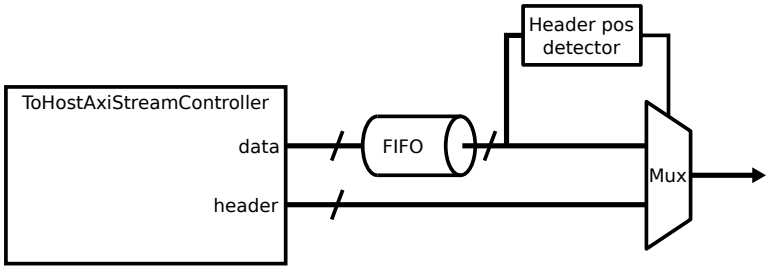


Figure 18: A possible method of inserting headers into the output.

This solution, while already slightly convoluted, has a number of problems. One problem is that the FIFO can hold more than one chunk at a time. Because of this, the headers would need te be stored in a second, smaller FIFO to make sure every chunk gets its corresponding header. Additionally, the output word width of the FIFO differs from its input word width. This makes inserting the header into the right position of the output data a difficult task.

5.1.2 A special FIFO

A better solution is to change the way the FIFO itself works, by making a special kind of FIFO. This special FIFO can be used as a normal FIFO, while also having the ability to reserve a word in its memory for a header. This reserved word can then be written to at a later time, when the contents of the header are known. An example of the writing process of such a FIFO is shown in fig. 19. This is the design that was chosen to be implemented. The next sections will describe the design of such a FIFO.

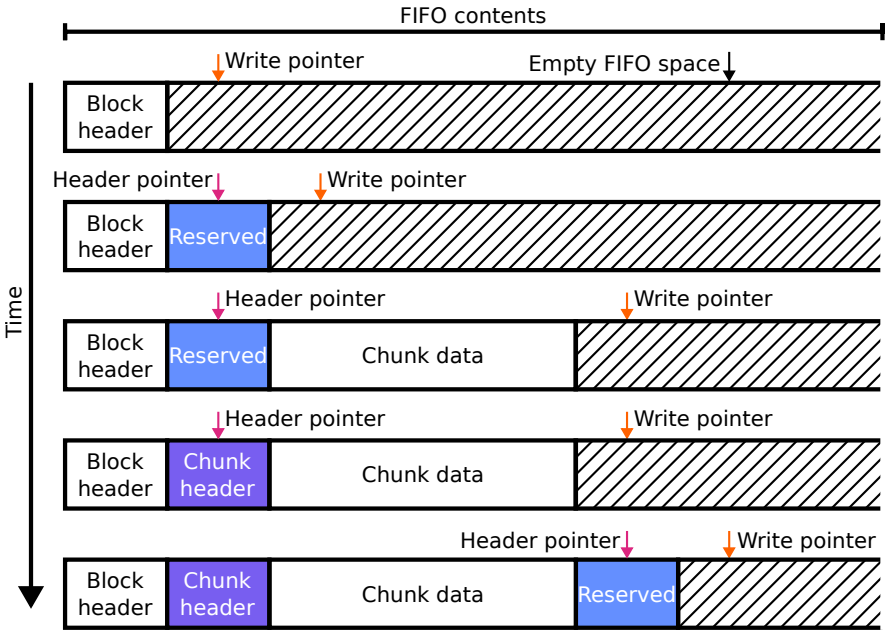


Figure 19: An example of writing chunk data to a FIFO with header insertion.

5.2 FIFO

To implement the special FIFO, a normal FIFO must be implemented first. After implementing a normal FIFO, it can be adjusted to have a header insertion functionality. A FIFO (First In First Out) is a component that can store data. It can be written to and read from, possibly in different clock domains and with differing word sizes. All data that enters a FIFO will exit it in the same order. A diagram of the inputs and outputs of a FIFO is shown in fig. 20. On the writing side it has a data input (din), a write enable input (wr_en), a full output and a write clock input (wr_clk). On the reading side it has a data output (dout), a read enable input (rd_en) and an empty output. If the FIFO is asynchronous, meaning reading and writing are done in separate clock domains, the reading side also has its own read clock (rd_clk).

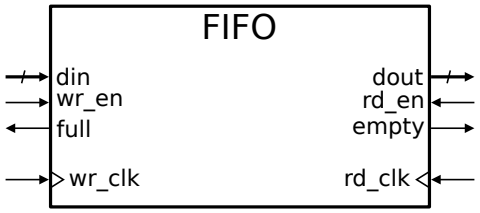


Figure 20: The inputs and outputs of a FIFO.

A FIFO within an FPGA utilizes a memory block to store the incoming data. There are synchronous and asynchronous types of memory blocks. Because the CRToHost FIFO will be read from and written to using different clocks, an asynchronous memory block is used. The CRToHost will also require differing word sizes for the read and write data.

Figure 21 shows a generalized diagram of a typical FIFO using a memory block. The memory block is written to whenever the write enable pin (wr_en) is asserted, after which the writing address is incremented. To read from the FIFO, the read enable pin (rd_en) needs to be asserted. The FIFO will then output the next word on the next rising edge of the clock. It will keep incrementing the reading address on every clock cycle as long as rd_en is asserted.

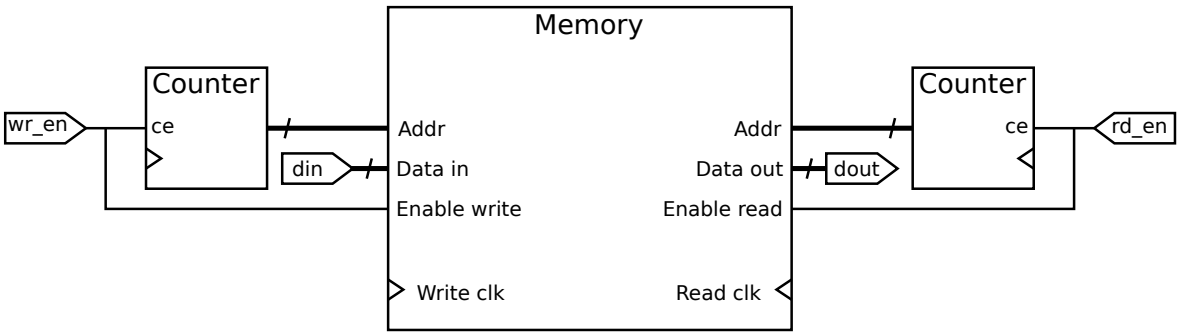


Figure 21: A typical FIFO. 'ce' stands for clock enable; when it is asserted the counter increments its output by one on every clock cycle.

The memory addresses need to wrap around to the beginning once the end of the available memory is reached. This can be done very easily by using a block of memory with a size equal to a power of two. This way, the memory addresses will automatically wrap around back to zero when they are incremented past their maximum value.

5.2.1 FIFO with a header inserting functionality

Now that it is clear how a normal FIFO functions, the functionality of a header-inserting FIFO can be discussed. As was described in section 5.1.2, this FIFO needs to reserve a space for a chunk header, so that it can be written to at a later time. Figure 22 shows a write sequence into this special FIFO. Data words marked with A, B, C, H, and D are written into the FIFO. When B is written, new_chunk is asserted. This tells the FIFO that a new chunk is starting, which means it needs to reserve a word of data in between A and B for a chunk header. After writing C, H is written to the FIFO with set_header asserted. This should write H into the reserved header position between A and B. Figure 23 shows the resulting contents of the FIFO.

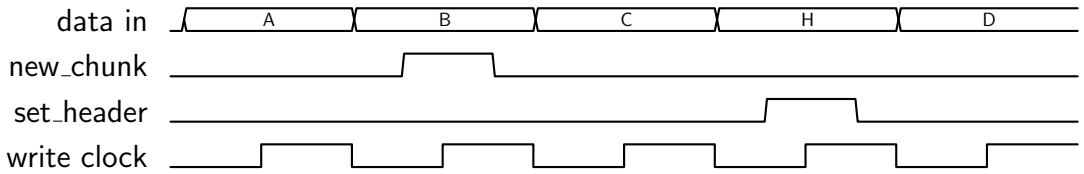


Figure 22: The process of writing data into the special FIFO. The write enable signal is asserted during the entire sequence.

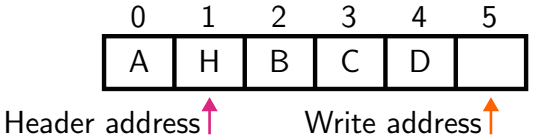


Figure 23: The contents and of the FIFO after the writing sequence shown in fig. 22.

A normal FIFO will increment its writing address by one on every clock cycle the write enable signal is asserted. The header inserting FIFO does exactly the same as long as new_chunk and set_header remain de-asserted. Whenever new_chunk is asserted during a clock cycle, the FIFO needs to reserve a word in its memory for a chunk header. The word being written when new_chunk is asserted (B in figs. 22 and 23) is the first data word of the chunk. Thus, the this word must be written to one address after the one it would have normally been written to. This can be done by adding additional logic to the write address counter shown in fig. 21.

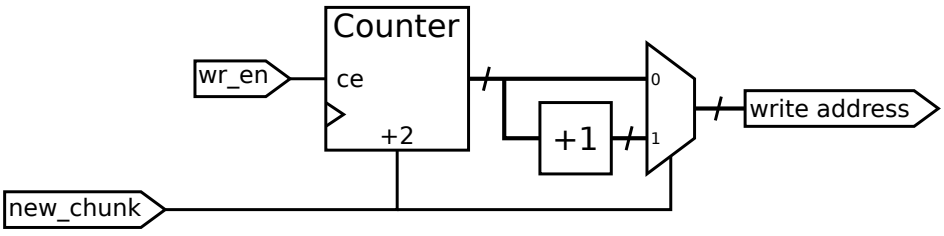


Figure 24: The logic responsible for reserving a word of data in the memory of the FIFO.

Figure 24 shows a logic setup that can achieve this. It incorporates a counter similar to the ones in fig. 21, however this counter has an additional '+2' input. When this input is asserted during a clock cycle, the counter increments its output by two instead of one. The

multiplexer at the output of the setup switches between the output of the counter and the output of the counter plus one. This sets the writing address to the correct value as soon as new_chunk is asserted, without waiting for the next clock cycle. This ensures that the word being written when new_chunk is asserted is placed after the reserved word, rather than in front of it.

This setup does not have the functionality to be able to write to the reserved word after it was created. For this, the address of the reserved word must be stored. This can be done with a register connected to the output of the counter. The clock enable pin of this register should be connected to new_chunk. This way, it will only store the output of the counter whenever new_chunk is asserted, which will be equal to the address of the reserved word.

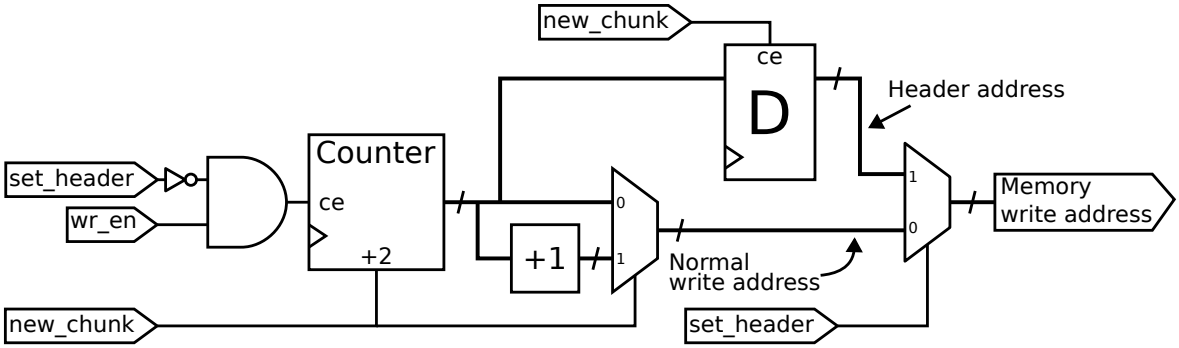


Figure 25: The write pointer logic within the header-inserting FIFO.

Figure 25 shows the final design for the write pointer logic. This design replaces the writing counter in fig. 21. As fig. 25 shows, the output is switched between the header address and the normal write address by the set_header input. This switches the memory writing address to the address of the reserved word when set_header is asserted, so that the reserved header position can be written to. The normal writing address should not be incremented when this happens, as this would leave gaps in the memory. For this reason, the clock of the counter is only enabled when wr_en is high and set_header is low.

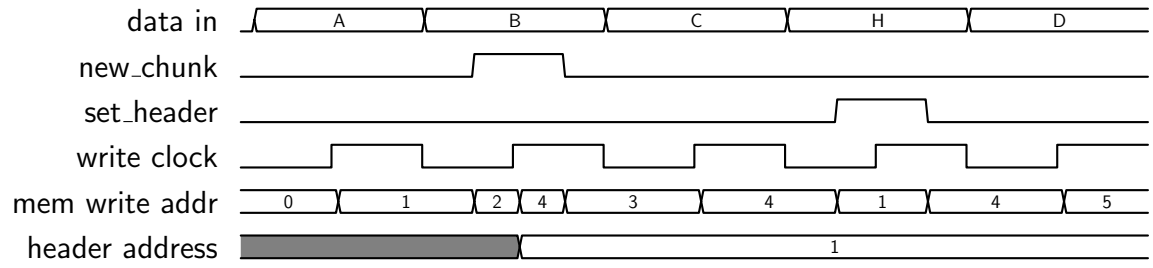


Figure 26: The process of writing data into the special FIFO. The memory write address that results from the logic described in this section is shown as well, in the 'mem write addr' row.

The resulting write sequence is shown in fig. 26. This sequence shows the memory write address during the same write sequence as shown in fig. 22. In this sequence, the memory write address temporarily changes to 4, before falling back to 3 when new_chunk is de-asserted. This is caused by the multiplexer placed after the counter, and will not cause any problems as long as new_chunk is only ever asserted for 1 clock cycle.

5.2.2 First-word fall-through

The FIFO used by the CRToHost data manager is a first-word fall-through FIFO. This means that, instead of only outputting data after rd_en is asserted, the FIFO will always output the data pointed to by the read pointer. Asserting rd_en will tell the FIFO to start incrementing the read pointer. To implement this, the reading address must be incremented as soon as rd_en is asserted. This can be done by using a multiplexer, as shown in fig. 27. Figure 28 shows the resulting time diagram.

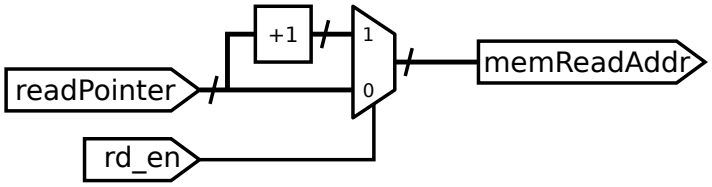


Figure 27: The first-word fall-through memory reading logic.

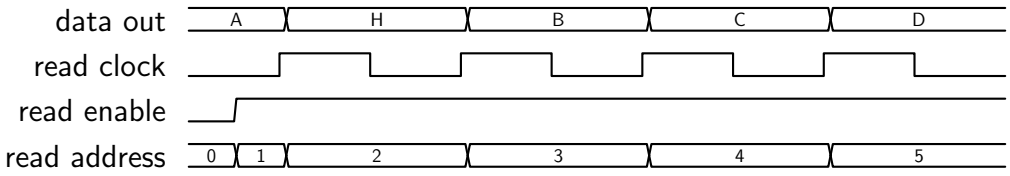


Figure 28: Reading the data that was written in fig. 22 from the special FIFO with first-word fall-through.

5.2.3 The empty and full outputs

The FIFO has an empty and a full output. The empty output is used to determine if there is any data left to read. The read pointer will not be incremented as long as the empty output is asserted. The full output is used to determine if there is any more space to write data into. The write pointer will not be incremented as long as the full output is asserted.

The state of these outputs is determined by the positions of the read and write pointers. As is shown in fig. 29, the FIFO is considered full when the write pointer is positioned at the same word as the read pointer. Figure 29 also shows that the FIFO is considered empty in two situations: when the read pointer is positioned one word before the write pointer, and when it is positioned one word before the header pointer when the header is not yet written to. This last situation is to prevent the read pointer from passing an unwritten header, which would result in the unwritten header being outputted erroneously.

Thus, to determine the state of both the full and empty outputs, the read, write and header pointers need to be compared to each other. This poses a problem, because these pointers are stored in separate clock domains.

5.2.4 Clock domain crossing

A clock domain is a group of logic that is clocked by the same clock signal. In the case of the CRToHost block, the input is clocked by a multiple of the 40 MHz LHC clock used by ATLAS, while the output is clocked by the 250 MHz PCIe clock [5]. This means that the

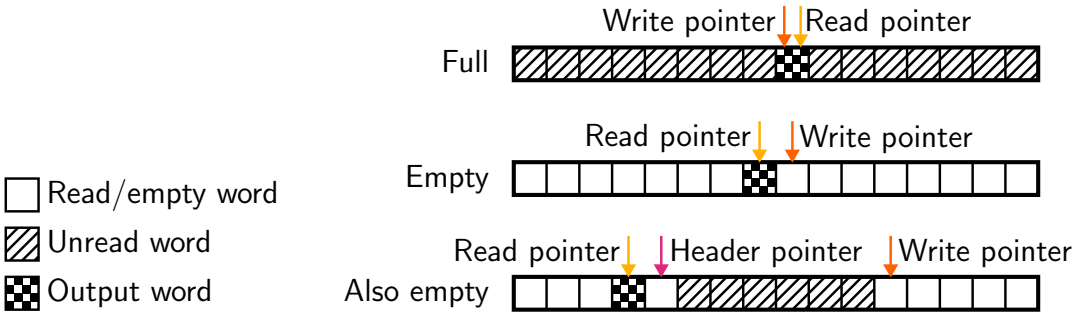


Figure 29: The contents of the FIFO when it is full or empty. The word size of the reading side is the same as the writing side for simplicity.

incrementation of the writing pointer is not synchronized to that of the reading pointer. Trying to compare these two values directly can cause the output of the register storing the result to become metastable. Metastability is caused by a change on the input of a register during its setup time, which is explained further in section 5.3. A solution to this problem is clock domain crossing (CDC). Clock domain crossing is done by connecting the signal that needs to cross clock domains to a minimum of two cascaded D-flipflops, as shown in fig. 30. This causes the output to follow the input signal, with a delay of a minimum of two clock cycles of the destination clock (clk 2 in fig. 30).

The output will not contain every sample of the input, especially when the output clock is slower than the input clock. This is not a problem for the empty and full signals, however. While it is critical for the full and empty outputs to be asserted in time, a delayed de-assertion will not cause any problems, aside from slowing down the writing or reading speed. Because the full output resides in the writing clock domain, only the read pointer is crossed from another clock domain. This means that the read pointer signal could be delayed, and some samples might be lost due to the CDC. This could delay the de-assertion of the full signal, but will never hinder its assertion, thus causing no critical problems. The same holds for the empty signal, as the only signals that need to be crossed are the write pointer and the header pointer.

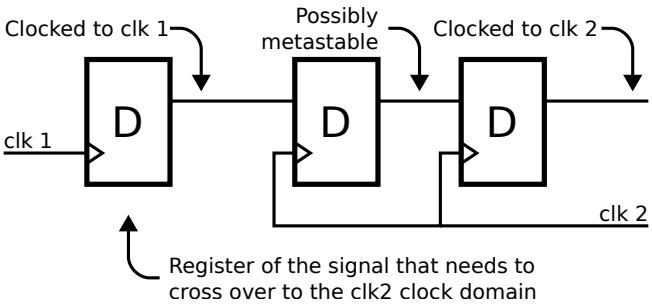


Figure 30: How clock domain crossing is done.

This method of clock domain crossing only works for single-bit signals. When the input of the first register changes at the wrong moment, its output could become metastable. The metastable output will eventually settle on either high or low, which is acceptable for single-bit signals. When synchronizing a multi-bit signal however, some flipflops could settle on the correct value on later clock cycles. This causes the output to become a value that was never on the input, which is not acceptable.

A solution for this is to use the Gray code. The Gray code is a way of encoding integers in binary format, such that successive numbers differ in only one bit [7]. This means that, when a given integer is represented in Gray code, crossing it over to a different clock domain will not cause any problems, as long as it is only ever incremented or decremented by one. This is true for the read and write pointers, making the use of the Gray code the perfect method of clock domain crossing for these signals. The read and write pointers are not represented in Gray code, however. They will need to be converted to Gray code before crossing to the other clock domain, and then converted back to binary afterwards. The resulting block diagram is shown in fig. 31.

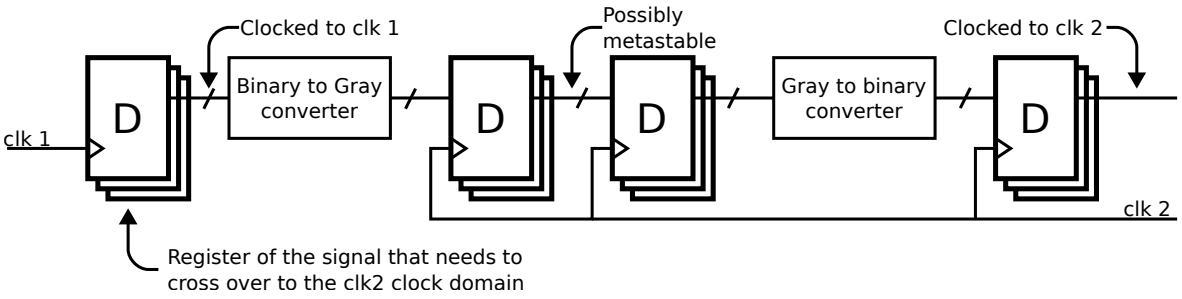


Figure 31: How clock domain crossing is done for multi-bit values by using the Gray code. This only works if the integer in question is only ever incremented or decremented by one.

5.2.5 ToHostAxiStreamController implementation

To be able to use the new FIFO, the ToHostAxiStreamController needs to be adjusted. It uses the toBlock process to convert the incoming stream of data into chunks within blocks. The only thing that needs to change about this process is that it needs to assert the new_chunk and set_header signals of the special FIFO at the appropriate times.

The toBlock process keeps track of the length of the chunk it is creating in a counter register. This value can be used to assert the new_chunk signal. It should be asserted whenever the chunk length is 0 and toBlock is not creating a block header.

When it has finished creating a chunk, the original toBlock process generated a trailer for it. The signal that activates the trailer generation can simply be connected to set_header. This way, the trailer it generates is moved to the start of the chunk by the header-inserting FIFO, making it a header.

This results in the diagram shown in fig. 32.

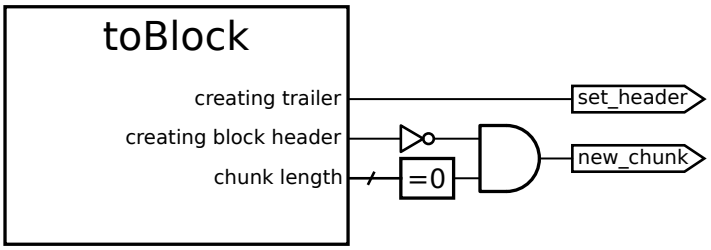


Figure 32: A very abstracted diagram of how the toBlock process asserts new_chunk and set_header.

5.3 Timing requirements

After synthesizing the design for the special FIFO described in the previous section (section 5.2.1), it does not pass the timing requirements. Timing is a very important aspect to keep in mind when designing firmware for an FPGA. Signals have a limited amount of time to travel from the output of one flipflop to the input of the next. The time this takes is called the propagation delay (t_p). The propagation delay can be influenced by both the distance a signal has to travel and the amount of logic it has to pass.

The signal at the input of a flipflop also needs to be stable for a certain amount of time before the rising edge of the clock, which is called the setup time (t_s).

Both of these values are pictured in fig. 33. By subtracting the setup time from the clock period, the maximum propagation delay can be found. When a signal exceeds this maximum, the output of the receiving flipflop can become metastable, which is undesirable.

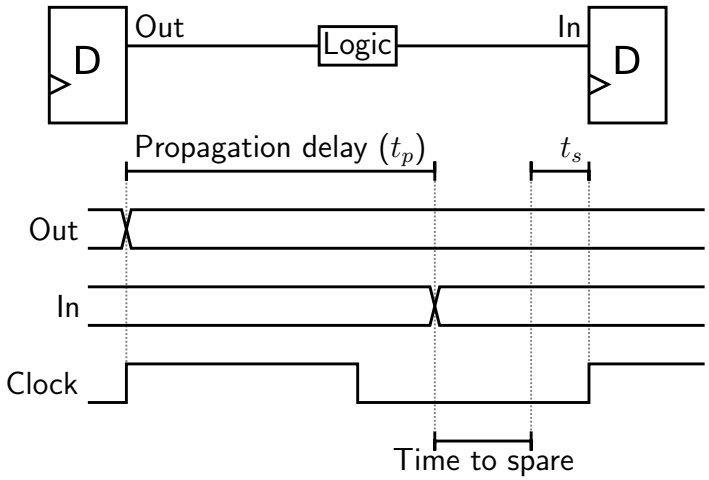


Figure 33: A diagram to clarify the meaning of propagation delay and setup time.

The timing requirement severely limits the complexity of operations that can be done to a set of signals within a single clock cycle. Some operations, such as addition [8], take a relatively long time to complete. This makes the use of intermediate flipflops necessary when doing larger calculations.

5.3.1 Pipelining the writing signals

As stated in section 5.3, the design for the special FIFO does not pass the timing requirement tests. The test results state this is caused by the memory writing signals. The reason for this is most likely the addition done by the counter block in fig. 24, in combination with the addition done by the '+1' block. These operations take a significant amount of time to complete. This, coupled with the distance between CRToHost and the memory block, causes the propagation delay of the memory write address signal (memWriteAddr) to exceed its maximum.

To resolve this, flip-flops are used to buffer the writing signals before they arrive at the memory block, as shown in fig. 34. This is also called 'pipelining', and reduces the distance the signals have to travel within one clock cycle. A delay of one clock cycle will be introduced to the writing process, which will also delay the de-assertion of the 'empty' output signal by one clock cycle. This will, however, not harm the operation of the FIFO.

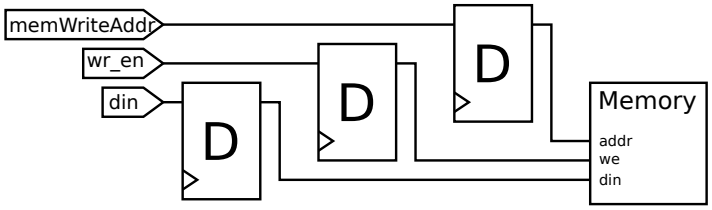


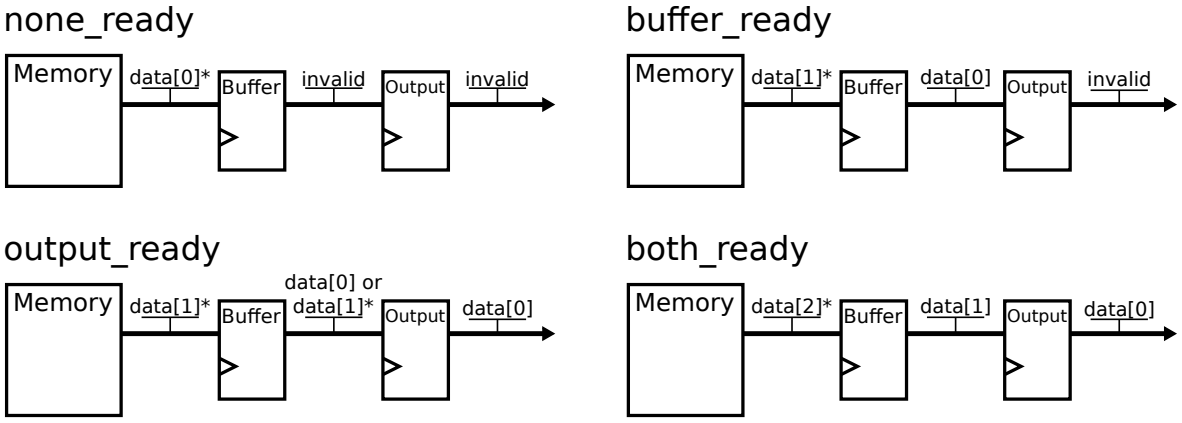
Figure 34: The memory writing signals, pipelined through flip-flops to account for the timing problems.

5.3.2 Pipelining the output

After implementing the pipelines described in section 5.3.1, the timing requirements are still not met. The output of the FIFO takes too long to reach the input of the next system (in this case the multiplexer at the output of CRToHost, as shown in fig. 17).

Resolving this problem requires a more complex solution than just pipelining the output through a flipflop. This is because the FIFO needs to be 'first-word fall-through', as described in section 5.2.2. This means that the data output must always have the first word available to be read, as long as the FIFO is not empty. Whenever rd_en is asserted, the output should change to the next word on the next rising edge of the clock.

To achieve this, while also conforming to the timing requirements, a state machine can be used. This solution is also used by Xilinx for their FIFOs. The state machine can be used to load the FIFO data into the output registers while the FIFO is being written to, so that the data is ready when the FIFO is read. The memory requires two output registers, one of which must be close to the memory block to be able to conform to the timing requirements. This results in four possible states, one for every combination of valid data being loaded into the registers. These states are shown in fig. 35.



* = might be invalid

Figure 35: The four different output states.

The state machine starts at the none_ready state, as both registers will contain invalid information after a reset. The next state will depend on the validity of the output of the memory block. The output is valid when the word on the current read address has been fully written to. When the memory is empty, the memoryEmpty signal is asserted.

Another factor is the rd_en signal. When the state machine is in either the output_ready or the both_ready state, the output is available to be read. When rd_en is high during a

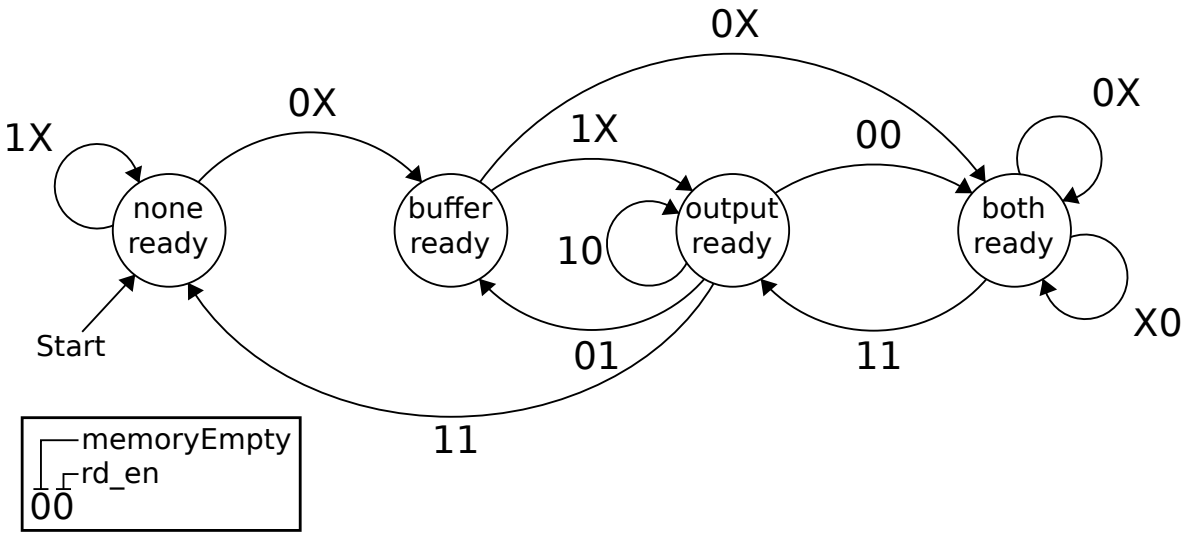


Figure 36: The output state machine.

rising edge of the clock in either of these states, the next data word must be outputted or the 'empty' output signal must be asserted. The status of the rd_en signal has effect on the next state. If for instance the rd_en signal is high in the output_ready state, the next output cannot be provided in time. This is because, in this state, the buffer register either still holds the previous word or an invalid version of the next one. The next state will need to be either none_ready or buffer_ready, depending on the status of the memoryEmpty signal. The full state machine diagram can be found in fig. 36.

The state of the state machine controls a number of signals. Among these signals are the clock enable pins for the buffer and the output register. These need to be controlled independently, so as to not lose any data. The 'empty' output signal is de-asserted when the state is either output_ready or both_ready, as those states indicate data being ready at the output. Finally, the readCounter process, responsible for incrementing the read pointer, is also controlled by the current state. Figure 37 shows the inputs and outputs of the state machine in a diagram.

After implementing the improvements described in this section, the timing requirements are met.

5.4 Firmware

A simplified version of the final design of the header-inserting FIFO can be seen in fig. 38. This design has been implemented into VHDL firmware. Because the VDHL implementation is essentially the described design written in a hardware description language, no separate implementation chapter is provided.

The final synthesized firmware design uses 167 lookup tables and 236 registers. The original design, using a standard Xilinx library FIFO, used 231 lookup tables and 278 registers. Thus, the resource specification (specification 2 in section 4) has been reached. Because the header-inserting FIFO has the same size as the normal FIFO it replaces, it does not use any more memory than the original design, thus also fulfilling specification 3.

The firmware that was made for the header inserting FIFO can be found in appendix A. It makes use of libraries written by Xilinx for the implementation of the memory block, CDC, and Gray code conversion.

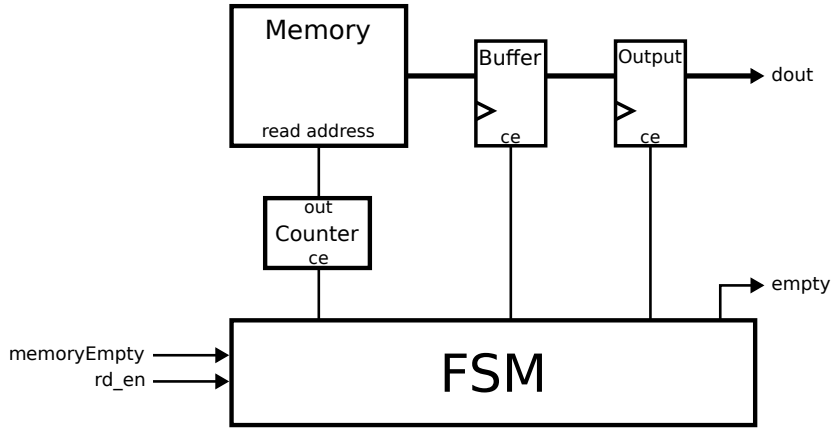


Figure 37: A diagram of the memory output within the FIFO. The FSM block in this figure contains the state machine pictured in fig. 36. 'ce' stands for clock enable, and functions as the name implies.

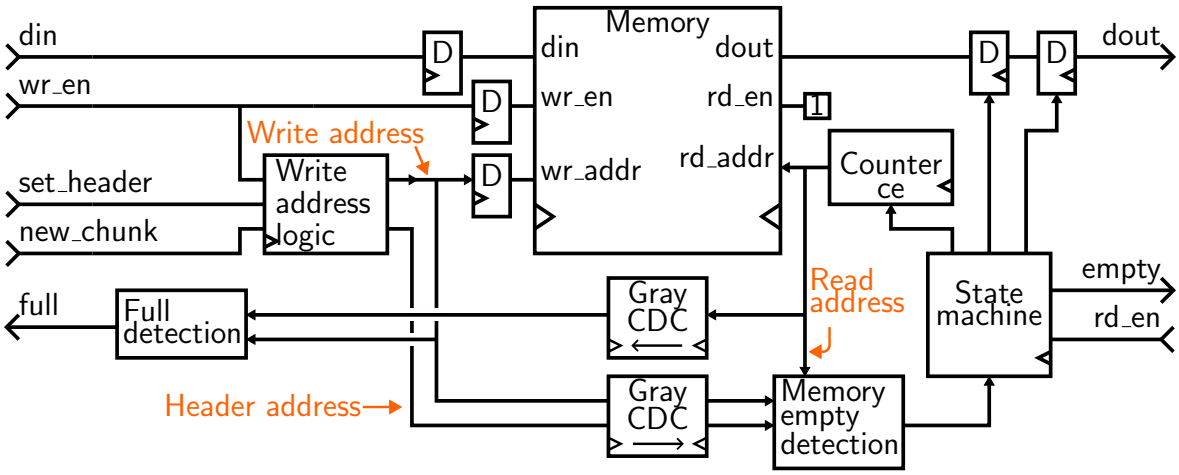


Figure 38: A simplified block diagram of the complete firmware design for the header-inserting FIFO.

5.5 Software

The FELIX software reads the data that was put into the CMEM buffer, decodes it, and publishes it to the network. The decoding part of the software needs to be adjusted to be able to handle chunk headers.

The FELIX software is written in C++ and uses a decoder class to handle the decoding of blocks. Every ELink has a separate decoder object, which among other data stores the last subchunk of the previous block, the amount of data it has processed, and the block sequence number (shown in fig. 11a of section 3.3.1). Every time the FELIX software reads a new data block, it delegates the decoding of it to its respective decoder object.

The decoder objects follow a relatively simple procedure for decoding. They first go through the entire block from back to front, using the chunk trailers, while storing the index of the start of every chunk in an array. Subsequently, they iterate through this array, publishing each (sub)chunk from front to back.

For the software to be able to read a format that uses chunk headers, the first step becomes unnecessary. The second step needs to be adjusted to read chunk headers, rather

than reading indices from the array that the first step provided. Making these changes to the software is a very simple task, and does not require much time. The result of the change can be found in appendix B, which contains a code listing with the adjusted decoder function.

6 Results

Multiple tests can be done to verify the implementation of the chunk header generation. The first property to be tested is the generated chunks themselves. These need to follow the proper format and not lose any data. Next, the efficiency of the new format can be tested, to verify that it is indeed more efficient than the original format.

6.1 Verification test

To verify that the design outputs the correct data format, the following materials are required:

- A server with an available PCIe slot and the CMEM driver installed.
- The FELIX software, which can be found in the felix-distribution repository within the atlas-tdaq-felix group on the CERN GitLab server.
- A FELIX PCIe card, with the firmware that needs to be tested installed on it.

To test the data output of the FELIX card, it will need a data input of some sort. Because ATLAS detectors are usually not readily available, the data normally produced by the front-ends must be simulated. The FELIX firmware comes equipped with an internal emulator for this purpose, as shown in fig. 39. The emulator behaves differently based on the FELIX flavour (FELIX flavours were described in section 3.1.1). In GBT mode, the emulator outputs data into the decoding block, which then converts it into AXI streams before entering CRToHost. In FULL mode, the emulator skips the decoder, and outputs its data directly into CRToHost, as an AXI stream. The FULL mode emulator also supports being triggered by TTC[6] triggers, which means it can also be triggered by the TTC emulator. This is useful, because the TTC emulator can easily be configured to generate triggers at any frequency.

6.1.1 Performing the test

The verification test is done with FULL mode firmware. If the adjusted CRToHost block functions correctly in FULL mode, it can be assumed that it will function in GBT mode as well. To collect the data the FELIX card produces, the data emulator needs to be activated. This is done by using the TTC emulator, which allows for easy frequency adjustments.

To perform the verification test, a number of programs are used from the FELIX software. More information about these programs can be found in [2]. The FELIX card is configured using the `elinkconfig` program. After this, the `fttcemu` program is used to activate the TTC emulator and set a trigger rate of a certain frequency. The `femu` program is used to configure the internal emulator to accept TTC triggers. Next, the `fdaq` program is used to directly write the incoming data into a file. The file can be analysed by the `fcheck` program, which was adjusted to be able to accept and verify data files with chunk headers.

[6] Timing, Trigger and Control (TTC) is a system used at CERN to trigger the detector electronics in sync with particle collisions [9].

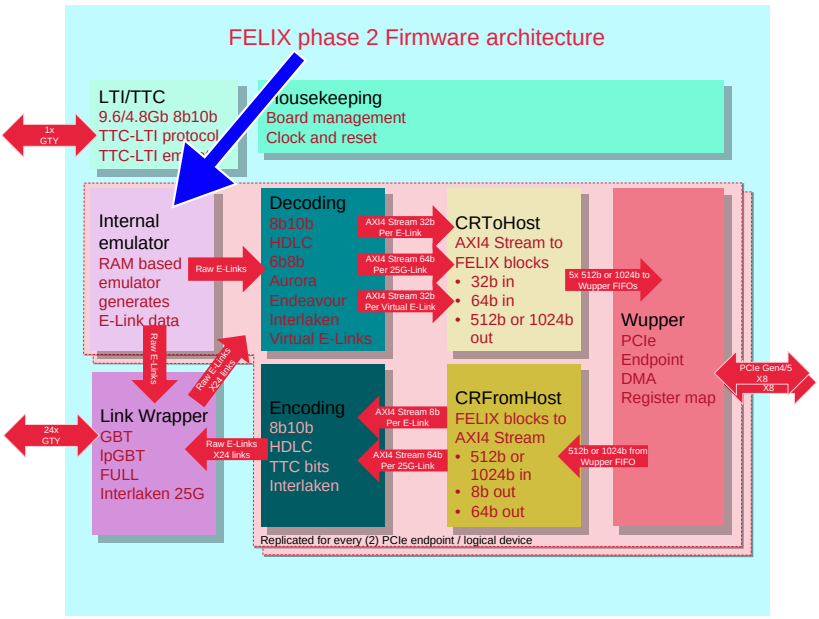


Figure 39: The FELIX firmware diagram from fig. 4, with a large arrow pointing to the internal emulator [5].

6.1.2 Results

The data was generated successfully, after which it was checked with the `fcheck` program. This program has two useful output modes. The first output mode checks the entire data file for problems, and outputs these problems. The second output mode outputs the raw data words of a given number of blocks.

The program reported zero problems when run in the first output mode. It can thus be assumed that the data generated by the implementation follows the correct format. Running the program with the second output mode resulted in the output shown in listing 1. This listing shows the raw data that was received by the FELIX server. This data shows that the implementation has successfully placed a chunk header (`60000020`) directly after the block header (`c0ce0040`), and in front of every subsequent chunk.

The chunk trailer format in fig. 11b from section 3.3.1 specifies that the least significant two bytes of a chunk trailer indicate the length of the chunk. The chunk headers follow the same format. All but one of the headers in listing 1 end with `0x0020`, indicating that every one of these headers has a chunk of 32 bytes following it. This corresponds to the amount of four byte words following each header, meaning that the chunk headers accurately reflect the information about their respective chunks.

The last chunk header (`20000008`) is different from the rest, as its chunk is cut off by the end of the block. It was divided into two subchunks, one of which will have been placed in the next data block of this link. The fact that this is a subchunk is indicated by the `0x20` at the start of the chunk header. The last two bytes of the trailer indicate a length of 8 bytes instead of the usual 32, which correctly corresponds to the two four byte data words following the trailer.

```
$ fcheck chunkheader_datafile.dat -H -c -4 -F 1
=> File: chunkheader_datafile.dat
Blocksize: 1024
==> BLOCK 0 (E=040=1-0 seq=0):
```

```
   0:  c0ce0040 60000020 001800aa 10aabb00
  16:  03020100 07060504 0b0a0908 0f0e0d0c
  32:  13121110 17161514 60000020 001800aa
  48:  10aabb01 03020100 07060504 0b0a0908
  64:  0f0e0d0c 13121110 17161514 60000020
  80:  001800aa 10aabb02 03020100 07060504
  96:  0b0a0908 0f0e0d0c 13121110 17161514
 112:  60000020 001800aa 10aabb03 03020100
------------ 48 lines omitted ------------
 896:  13121110 17161514 60000020 001800aa
 912:  10aabb19 03020100 07060504 0b0a0908
 928:  0f0e0d0c 13121110 17161514 60000020
 944:  001800aa 10aabb1a 03020100 07060504
 960:  0b0a0908 0f0e0d0c 13121110 17161514
 976:  60000020 001800aa 10aabb1b 03020100
 992:  07060504 0b0a0908 0f0e0d0c 13121110
1008:  17161514 20000008 001800aa 10aabb1c
File contains 112929792 bytes (110283 FLX blocks)
```

Listing 1: The first lines of the output of the `fcheck` program with options enabled to list the first data words within the data file. Block headers are colored purple, chunk headers are colored orange.

6.1.3 Discussion

The verification test resulted in the FELIX card outputting valid data in the correct format. Every (sub)chunk has a header in front of it, each of which contains the correct information about its chunk. Thus, the header-inserting FIFO implementation is working as intended.

6.2 Efficiency test

The goal of this project was to improve the efficiency of the FELIX server software, specifically the toHost program, by changing the data format it needs to parse. The efficiency of a program can be measured in numerous ways, the most obvious of which is to measure the maximum data rate the program can process. Measuring the maximum data rate is difficult, however. It is possible to keep increasing the trigger frequency until the incoming data stops being parsed correctly, but the frequency at which this happens will vary every time the test is done. A better way of measuring the efficiency of the implementation is to measure the CPU usage the toHost program requires at a certain trigger frequency. This way, the old implementation that uses chunk trailers can be compared to the new one, which uses chunk headers.

This test requires the following materials:

- A server with an available PCIe slot and the CMEM driver installed. This server is called the FELIX server.
- Another server with a 100 Gigabit Ethernet (100 GbE) connection to the first server. This server is called the data handler server.
- The FELIX software installed on both servers, which can be found in the felix-distribution repository within the atlas-tdaq-felix group on the CERN GitLab server.
- A FELIX PCIe card, with the firmware that needs to be tested installed on it.

The flavour of the firmware is important to consider for this test. FULL mode produces relatively large chunks from very few ELinks, while GBT mode produces smaller chunks from a large number of ELinks. This will have an effect on the efficiency of the toHost program, so it is best to perform the tests twice, with both firmware flavours.

6.2.1 Performing the test

To perform the test, the toHost program needs to be started on the FELIX server. The data emulator should be configured to accept triggers from the TTC emulator. The TTC emulator frequency needs to be set to 0 Hz.

The data handler server should be running the `test-swrod` software, which collects the data from the FELIX server through the 100 GbE connection. The FELIX server should be running the `FELIX-toHost` program. It may not run any other programs.

To collect a reasonable amount of samples, a script was made to measure the average CPU usage of the toHost program over one second. After each measurement, 1 kHz is added to the TTC emulator frequency. This is done from 1 kHz to 1 MHz.

6.2.2 Results

The efficiency test results are shown in figs. 40 and 41.

Figure 40: The results of the efficiency test with FULL mode firmware.

6.2.3 Discussion

The results of the GBT mode test, which can be seen in fig. 41, show a significant decrease in CPU usage when chunk headers are used. The proportional decrease in CPU usage is

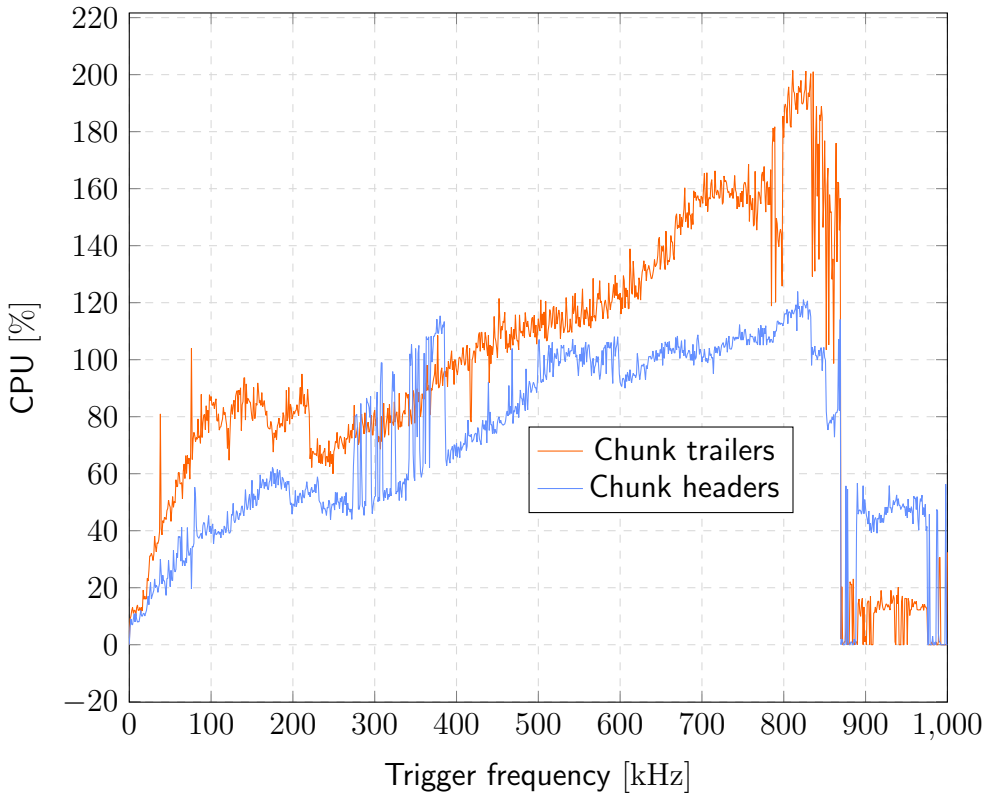


Figure 41: The results the efficiency test with GBT mode firmware. From 0 kHz to 800 kHz the average proportional decrease in CPU usage from chunk trailers to chunk headers is 26.8 %.

26.8 %, which is more than the amount achieved in the simulation done by Kolos, described in section 3.2 [4]. There are multiple possible reasons for this, one of which is that the simulation software is not identical to the FELIX-ToHost software, which could affect the resulting efficiency.

The results of the GBT test show a downwards jump at approximately 875 kHz. This can be explained by the maximum data rate the ELinks can support. The chunk length during this test was set to 34 bytes. The format used by the GBT emulator was the 8bit/10bit format, which sends 8 bits of useful data in a word of 10 bits, with 2 extra bits to ensure a good DC balancing and to signal out-of-band data characters. This means that, for 34 bytes, 340 bits are sent for every chunk. The 8bit/10bit format also requires 2 additional bytes per chunk for start-of-packet and end-of-packet metadata. With 10 bits per byte this adds up to a total of 360 bits per chunk. The ELink was configured as an 8-bit ELink which means it can only process 8 bits every clock cycle. With a clock frequency of 40 MHz, this results in a maximum chunk processing rate of 889 000 chunks per second. This is most likely the reason for the dip in the graph in fig. 41, which appears to happen at that frequency.

The FULL mode test, shown in fig. 40, does not show any apparent improvement. As of right now the reasons for this are unclear. The new format does, however, appear to keep working for longer than the original one.

7 Conclusion

The goal of this thesis was to determine whether the efficiency of the FELIX software can be significantly improved by changing the ToHost data block format, and if so, to implement the new format.

To answer the first two research sub-questions from section 2, the efficiency improvement a format offers must be determined. To do this, simulations were done. Three formats were proposed, two of which were simulated. Compared to the original format, the first format was 2.8 % faster with smaller chunk sizes and 38.2 % faster with larger chunk sizes. The second format was 18.5 % faster with smaller chunk sizes and 37.9 % faster with larger ones. The third format was not simulated because of time constraints. Because the second format is an expansion of the first format, the first format was implemented first. The second format was not implemented due to a lack of time.

The new format was successfully implemented into the firmware, thereby addressing the third research sub-question from section 2. The synthesized firmware uses less resources than the system it replaces, which means that specification 2 in section 4 has been met. It uses an equal amount of memory, thus also completing specification 3.

The implementation produces the correct data format, and the software was modified to parse said data format. This completes specifications 1 and 4 in section 4. Its efficiency was measured by measuring the CPU usage of the software running on the server. In one firmware flavour, while it does keep working for higher frequencies, it is not any faster than the original firmware. However, in another firmware flavour, it is 26.8 % faster on average. This means that, for one firmware flavour, the efficiency specification of a minimum of 20 % has been reached (specification 5 in section 4). It is as of yet unknown why it is not faster in the other firmware flavour.

The implementation shows a slightly different improvement compared to the simulation. This could be related to the fact that the conditions of the simulation and the test of the implementation differ in terms of number of ELinks and the size of chunks. Both do, however, show a significant improvement.

Every sub-question has been answered and every specification has been met, which means the goal of this thesis was achieved.

8 Recommendations

After the implementation of the chunk header format, multiple recommendations can be made. These recommendations include further research that can be done, as well as further tests that can be performed.

8.1 Further testing

In the results of the efficiency test, described in section 6.2, the software did not show any improvements in efficiency when reading data from a FULL mode FELIX card. Reading from a GBT mode card did show a significant improvement however. Research will have to be done in order to find out what causes this discrepancy.

Furthermore, the efficiency tests can be repeated with different constant (or perhaps varying) data chunk lengths, which could produce different results.

8.2 The blockless format

The theoretical blockless format, proposed in section 3.3, needs to be researched further. It shows a significant increase in efficiency, but would also require a large amount of changes to be made to both the firmware and software.

8.3 The hybrid format

The hybrid format, described in section 3.4, needs to be simulated, in order to determine if it is any more efficient than the other formats have shown to be. If this is shown to be true, it could prove an easier to implement alternative to the blockless format, which still uses the constant-length data blocks the software relies on.

References

[1] J. Pequenao, "Computer generated image of the whole ATLAS detector", 2008. [Online]. Available: https://cds.cern.ch/record/1095924.

[2] *FELIX User Manual*, 5-59-g5e18662, CERN, Feb. 2024. [Online]. Available: https://cernbox.cern.ch/remote.php/dav/public-files/gdGmWdNciJkLaR1/felix-user-manual.pdf.

[3] A. Paramonov, "FELIX: the Detector Interface for the ATLAS Experiment at CERN", 2021. DOI: 10.1051/epjconf/202125104006. [Online]. Available: https://cds.cern.ch/record/2814356.

[4] S. Kolos, *Performance Study of the Phase-II FELIX Front-End SW Prototype*, Jun. 2023. [Online]. Available: https://cernbox.cern.ch/remote.php/dav/public-files/gdGmWdNciJkLaR1/Phase-2-FELIX-Study-2.pdf.

[5] *ATLAS FELIX firmware Phase-II Upgrade: Firmware specifications*, AT2-DQ-ES-0006, CERN, Nov. 2023. [Online]. Available: https://cernbox.cern.ch/remote.php/dav/public-files/gdGmWdNciJkLaR1/FELIX_Phase2_firmware_specs.pdf.

[6] M. Joos, "A package for the allocation of contiguous memory on Linux systems", Feb. 2021. [Online]. Available: https://cernbox.cern.ch/remote.php/dav/public-files/gdGmWdNciJkLaR1/cmem_rcc.pdf.

[7] F. Gray, "Pulse code communication", 2 632 058, Mar. 1953. [Online]. Available: https://ppubs.uspto.gov/dirsearch-public/print/downloadPdf/2632058.

[8] S. Xing and W. W. Yu, "Fpga adders: Performance evaluation and optimal design", *IEEE Design & Test of Computers*, vol. 15, no. 1, pp. 24–29, 1998.

[9] B. Taylor, "Ttc distribution for lhc detectors", *IEEE Transactions on Nuclear Science*, vol. 45, no. 3, pp. 821–828, 1998. DOI: 10.1109/23.682644.

A Header-inserting FIFO firmware

The header-inserting FIFO implementation was written in a single vhdl file, which is provided in this appendix.

```
--! This file is part of the FELIX firmware distribution (https://gitlab.
   cern.ch/atlas-tdaq-felix/firmware/).
--! Copyright (C) 2001-2021 CERN for the benefit of the ATLAS
   collaboration.
--! Authors:
--!               Jochem Leijenhorst
--!
--!   Licensed under the Apache License, Version 2.0 (the "License");
--!   you may not use this file except in compliance with the License.
--!   You may obtain a copy of the License at
--!
--!       http://www.apache.org/licenses/LICENSE-2.0
--!
--!   Unless required by applicable law or agreed to in writing, software
--!   distributed under the License is distributed on an "AS IS" BASIS,
--!   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied.
--!   See the License for the specific language governing permissions and
--!   limitations under the License.


----------------------------------
--! Company:  Nikhef
--! Engineer: Jochem Leijenhorst
--!
--! Create Date:    26/03/2024
--! Module Name:    HIFIFO
--! Project Name:   FELIX
----------------------------------

library ieee;
    use ieee.std_logic_1164.all;
    use ieee.numeric_std.all;
    use ieee.numeric_std_unsigned.all;

    use work.FELIX_package.all;


library xpm;
    use xpm.vcomponents.all;

entity HIFIFO is

    generic (
        -- Common module generics
        FIFO_WRITE_DEPTH        : integer  := 2048;
        WRITE_DATA_WIDTH        : integer  := 32;
        READ_DATA_WIDTH         : integer  := 32;
        PROG_FULL_THRESH        : integer  := 10;

        --! DO NOT EDIT THESE
```

```
        FIFO_READ_DEPTH         : integer := FIFO_WRITE_DEPTH *
            WRITE_DATA_WIDTH / READ_DATA_WIDTH;
        WRITE_POINTER_WIDTH     : integer := f_log2(FIFO_WRITE_DEPTH-1) +
             1;
        READ_POINTER_WIDTH      : integer := f_log2(FIFO_READ_DEPTH-1) +
            1
    );

    port (
        rst             : in std_logic;

        wr_clk          : in std_logic;
        wr_en           : in std_logic;
        din             : in std_logic_vector(WRITE_DATA_WIDTH-1 downto
            0);
        full            : out std_logic;
        prog_full       : out std_logic;
        set_header      : in std_logic;
        new_chunk       : in std_logic;
        wr_data_count   : out std_logic_vector(WRITE_POINTER_WIDTH-1
            downto 0);

        rd_clk          : in std_logic;
        rd_en           : in std_logic;
        dout            : out std_logic_vector(READ_DATA_WIDTH-1 downto
            0);
        empty           : out std_logic;
        rd_data_count   : out std_logic_vector(READ_POINTER_WIDTH-1
            downto 0)

    );

end HIFIFO;


architecture Behavioral of HIFIFO is

    -- _wr means converted to the write clock domain.
    -- _rd means converted to the read clock domain.
    signal readPointer      : std_logic_vector(READ_POINTER_WIDTH-1
        downto 0);
    signal readPointer_wr   : std_logic_vector(WRITE_POINTER_WIDTH-1
        downto 0);
    signal readPointerPlus1 : std_logic_vector(READ_POINTER_WIDTH-1
        downto 0);
    signal writePointer     : std_logic_vector(WRITE_POINTER_WIDTH-1
        downto 0);
    signal headPointer_rd   : std_logic_vector(READ_POINTER_WIDTH-1
        downto 0);
    signal writePointerPlus1: std_logic_vector(WRITE_POINTER_WIDTH-1
        downto 0);
    signal writePointerPlus2: std_logic_vector(WRITE_POINTER_WIDTH-1
        downto 0);
    signal writePointer_rd  : std_logic_vector(READ_POINTER_WIDTH-1
        downto 0);
```

```
    signal newWriteAddr     : std_logic_vector(WRITE_POINTER_WIDTH-1
       downto 0);
    signal memWriteAddr     : std_logic_vector(WRITE_POINTER_WIDTH-1
       downto 0);
    signal headPointer      : std_logic_vector(WRITE_POINTER_WIDTH-1
       downto 0);

    -- Keep track of writing the header for setting the empty flag.
    -- We don't want the read pointer to pass the headPointer when the
       header is still empty.
    signal isHeaderSet      : std_logic;
    signal isHeaderSet_rd   : std_logic;

    signal rst_rd           : std_logic;

    signal full_s           : std_logic;
    signal prog_full_s      : std_logic;
    signal wr_data_count_s  : std_logic_vector(WRITE_POINTER_WIDTH-1
       downto 0);

    signal memWriteAddr_p1  : std_logic_vector(WRITE_POINTER_WIDTH-1
       downto 0);
    signal wr_en_p1         : std_logic;
    signal din_p1           : std_logic_vector(WRITE_DATA_WIDTH-1 downto
       0);

    -- FSM for the stages of the output
    -- The output is loaded into the register within the memory block
       whenever it is ready
    -- It is ready when the first memory output word has been completely
       written to
    type output_state is (none_ready, buffer_ready, output_ready,
       both_ready);
    signal state : output_state;
    signal nextState : output_state;

    signal memoryEmpty : std_logic;
    signal memoryGoingEmpty : std_logic;
    signal memoryReadEnable : std_logic;

    -- Activates the clk of the last output register of the memory.
    signal outputBuffer : std_logic;
    signal doReadCount : std_logic;

begin
    full_s <= '1' when writePointerPlus1 = readPointer_wr or rst = '1'
       else '0';

    full <= full_s;

    wr_data_count <= wr_data_count_s;
    rd_data_count <= writePointer_rd - readPointer;

    prog_full_s <= '1' when wr_data_count_s >= PROG_FULL_THRESH or full_s
        = '1' else '0';

    writePointerPlus1 <= writePointer + 1;
```

```
writePointerPlus2 <= writePointer + 2;

newWriteAddr <= writePointerPlus1 when new_chunk = '1' else
   writePointer;
memWriteAddr <= headPointer when set_header = '1' else newWriteAddr;


writeCounter: process(wr_clk)
begin
    if rising_edge(wr_clk) then
        if rst = '1' then
            writePointer <= (others => '0');
            headPointer <= (others => '0');
            isHeaderSet <= '1';
            prog_full <= '1';
        else
            prog_full <= prog_full_s;
            if set_header then
                isHeaderSet <= '1';
            end if;
            if wr_en = '1' then
                if new_chunk = '1' then
                    headPointer <= writePointer;
                    writePointer <= writePointerPlus2;
                    isHeaderSet <= '0';
                elsif set_header = '0' then
                    -- Increasing the write pointer when setting the
                       header would leave gaps in the memory.
                    writePointer <= writePointerPlus1;
                end if;
            end if;
        end if;
    end if;
end process;


readCounter: process(rd_clk)
begin
    if rising_edge(rd_clk) then
        if rst = '1' then
            readPointer <= (others => '0');
            readPointerPlus1 <= std_logic_vector(to_unsigned(1,
               READ_POINTER_WIDTH));
        else
            if doReadCount = '1' then
                readPointer <= readPointer + 1;
                readPointerPlus1 <= readPointerPlus1 + 1;
            end if;
        end if;
    end if;
end process;


writePipe: process(wr_clk)
begin
    if rising_edge(wr_clk) then
        memWriteAddr_p1 <= memWriteAddr;
```

```
            wr_en_p1 <= wr_en;
            din_p1 <= din;
            wr_data_count_s <= writePointer - readPointer_wr;
        end if;
    end process;


    -- The FIFO should be seen as empty when the read pointer is on the
       write pointer,
    -- and when it's on the the header pointer when the header hasn't
       been set yet.
    memoryGoingEmpty <= '1' when (
        doReadCount = '1' and (
            readPointerPlus1 = writePointer_rd
            or
            (readPointerPlus1 = headPointer_rd and isHeaderSet_rd = '0')
        )
    ) else '0';

    memoryEmptyDriver: process(rd_clk)
    begin
        if rising_edge(rd_clk) then
            if rst_rd = '1' then
                memoryEmpty <= '1';
            else
                -- Decide if we want to keep the empty output high.
                if memoryGoingEmpty = '1' or (
                    -- This is basically the same logic as
                    -- setting the memoryGoingEmpty signal a few lines
                       back.
                    memoryEmpty = '1' and (
                        readPointer = writePointer_rd
                        or
                        (readPointer = headPointer_rd and isHeaderSet_rd
                           = '0')
                    )
                ) then
                    memoryEmpty <= '1';
                else
                    memoryEmpty <= '0';
                end if;
            end if;
        end if;
    end process;


    -- There are 2 registers/flipflops on the output of the memory.
    -- Let's call them buffer and output:
    --
    -- memory-->[buffer]-->[output]--dout-->
    --
    -- The output register's clock is activated by regceb which is driven
        by outputBuffer.

    -- There is an output available whenever the output register has
       valid data.
    -- That's the case in the output_ready and both_ready states.
```

```
-- Empty should be asserted in all other states.
empty <= '1' when state = none_ready or state = buffer_ready else
    '0';

-- outputBuffer activates the output register's clock, which will
    copy the value from the buffer register
-- to the output register during the clock cycles in which
    outputBuffer is asserted.
-- This should only happen when we have valid data in the buffer
    register and the last output has been read.
outputBuffer <= '1' when state = buffer_ready or (state = both_ready
    and rd_en = '1') else '0';

-- doReadCount enables the counter, which increments the readPointer
    on every clock cycle.
-- The address should be incremented whenever the previous value is
    valid (memoryEmpty = '0'),
-- and, if we're in the both_ready state, the last value has been
    read.
doReadCount <= '1' when memoryEmpty = '0' and not (rd_en = '0' and
    state = both_ready) else '0';

-- memoryReadEnable basically controls the clock of the buffer
    register.
-- It needs to activate whenever a new value is available to be read
    from the memory.
-- This is basically all the time, except when the last value has not
     been read yet in the both_ready state.
-- If we do enable the read in those conditions, a value will be lost
    , because the address has already been incremented.
memoryReadEnable <= '0' when state = both_ready and rd_en = '0' else
    '1';


FSM: process(state, memoryEmpty, rd_en)
begin
    case state is
        when none_ready =>
            -- The none_ready state indicates that there is neither
               valid data in the buffer nor the output register.
            if memoryEmpty = '0' then
                nextState <= buffer_ready;
            else
                nextState <= none_ready;
            end if;

        when buffer_ready =>
            -- buffer_ready means there is valid data available in
               the buffer register but not in the output register.
            -- outputBuffer is always 1 in this state.
            -- This means that we need to always leave this state
               immediately to not lose any data.
            if memoryEmpty = '0' then
                nextState <= both_ready;
            else
                nextState <= output_ready;
            end if;
```

```
            -- This is where rd_en starts mattering.
            when output_ready =>
                -- output_ready means there is invalid data in the buffer
                    register, but valid data in the output register.
                -- The output register is directly connected to dout,
                    which means that the FIFO can now be read from.
                if memoryEmpty = '0' then
                    if rd_en = '0' then
                        nextState <= both_ready;
                    else
                        nextState <= buffer_ready;
                    end if;
                else
                    if rd_en = '0' then
                        nextState <= output_ready;
                    else
                        nextState <= none_ready;
                    end if;
                end if;
            when both_ready =>
                -- both_ready means there is valid data in both the
                    buffer and the output register.
                -- This comfortable state is only left when both rd_en
                    and empty are asserted.
                if memoryEmpty = '1' and rd_en = '1' then
                    nextState <= output_ready;
                else
                    nextState <= both_ready;
                end if;
        end case;
    end process;

    FSM_set: process(rd_clk)
    begin
        if rising_edge(rd_clk) then
            if rst_rd = '1' then
                state <= none_ready;
            else
                state <= nextState;
            end if;
        end if;
    end process;


    memory: xpm_memory_tdpram
        generic map (

            -- Common module generics
            MEMORY_SIZE             => WRITE_DATA_WIDTH *
               FIFO_WRITE_DEPTH,
            MEMORY_PRIMITIVE        => "block",
            CLOCKING_MODE           => "independent_clock",
            ECC_MODE                => "no_ecc",
            MEMORY_INIT_FILE        => "none",
            MEMORY_INIT_PARAM       => "",
            USE_MEM_INIT            => 1,
```

```
        USE_MEM_INIT_MMI          => 0,
        WAKEUP_TIME               => "disable_sleep",
        AUTO_SLEEP_TIME           => 0,
        MESSAGE_CONTROL           => 0,
        USE_EMBEDDED_CONSTRAINT   => 0,
        MEMORY_OPTIMIZATION       => "true",
        CASCADE_HEIGHT            => 0,
        SIM_ASSERT_CHK            => 0,
        WRITE_PROTECT             => 1,

        -- Port A module generics
        WRITE_DATA_WIDTH_A  => WRITE_DATA_WIDTH,
        READ_DATA_WIDTH_A   => WRITE_DATA_WIDTH,
        BYTE_WRITE_WIDTH_A  => WRITE_DATA_WIDTH,
        ADDR_WIDTH_A        => WRITE_POINTER_WIDTH,
        READ_RESET_VALUE_A  => "0",
        READ_LATENCY_A      => 1,
        WRITE_MODE_A        => "write_first",
        RST_MODE_A          => "SYNC",

        -- Port B module generics
        WRITE_DATA_WIDTH_B  => READ_DATA_WIDTH,
        READ_DATA_WIDTH_B   => READ_DATA_WIDTH,
        BYTE_WRITE_WIDTH_B  => READ_DATA_WIDTH,
        ADDR_WIDTH_B        => READ_POINTER_WIDTH,
        READ_RESET_VALUE_B  => "0",
        READ_LATENCY_B      => 2,
        WRITE_MODE_B        => "read_first",
        RST_MODE_B          => "SYNC"

    )
    port map (

        -- Common module ports
        sleep          => '0',

        -- Port A module ports
        clka           => wr_clk,
        rsta           => rst,
        ena            => '1',
        regcea         => '1',
        wea            => (others => wr_en_p1),
        addra          => memWriteAddr_p1,
        dina           => din_p1,
        injectsbiterra => '0',
        injectdbiterra => '0',
        douta          => open,
        sbiterra       => open,
        dbiterra       => open,

        -- Port B module ports
        clkb           => rd_clk,
        rstb           => rst_rd,
        enb            => memoryReadEnable, --! yoo maybe do
            something with the reset here
        regceb         => outputBuffer,
        web            => "0",
```

```
        addrb           => readPointer,
        dinb            => (others => '0'),
        injectsbiterrb  => '0',
        injectdbiterrb  => '0',
        doutb           => dout,
        sbiterrb        => open,
        dbiterrb        => open
    );


-- Convert the reset to the read clock domain
cdc_reset: xpm_cdc_sync_rst
    generic map (
        DEST_SYNC_FF    => 2,
        INIT            => 1,
        INIT_SYNC_FF    => 0,
        SIM_ASSERT_CHK => 0
    )
    port map (
        src_rst  => rst,
        dest_clk => rd_clk,
        dest_rst => rst_rd
    );

-- Convert isHeaderSet to the read clock domain.
cdc_headerSet: xpm_cdc_single
    generic map(
        DEST_SYNC_FF    => 2,
        INIT_SYNC_FF    => 0,
        SIM_ASSERT_CHK => 0,
        SRC_INPUT_REG   => 1
    )
    port map (
        src_clk  => wr_clk,
        src_in   => isHeaderSet,
        dest_clk => rd_clk,
        dest_out => isHeaderSet_rd
    );


-- Convert the read pointer, write pointer and header pointer to
   their respective opposite clock domains.
checkMoreThan: if WRITE_POINTER_WIDTH < READ_POINTER_WIDTH generate
    cdc_headPointer: xpm_cdc_gray
        generic map (
            DEST_SYNC_FF           => 2,
            INIT_SYNC_FF           => 1,
            REG_OUTPUT             => 0,
            SIM_ASSERT_CHK         => 0,
            SIM_LOSSLESS_GRAY_CHK  => 0,
            WIDTH                  => WRITE_POINTER_WIDTH
        )
        port map (
            src_clk      => wr_clk,
            src_in_bin   => headPointer,
            dest_clk     => rd_clk,
```

```
            dest_out_bin => headPointer_rd(READ_POINTER_WIDTH-1
                downto READ_POINTER_WIDTH - WRITE_POINTER_WIDTH)
        );
    cdc_writePointer: xpm_cdc_gray
        generic map (
            DEST_SYNC_FF           => 2,
            INIT_SYNC_FF           => 1,
            REG_OUTPUT             => 0,
            SIM_ASSERT_CHK         => 0,
            SIM_LOSSLESS_GRAY_CHK  => 0,
            WIDTH                  => WRITE_POINTER_WIDTH
        )
        port map (
            src_clk      => wr_clk,
            src_in_bin   => writePointer,
            dest_clk     => rd_clk,
            dest_out_bin => writePointer_rd(READ_POINTER_WIDTH-1
                downto READ_POINTER_WIDTH - WRITE_POINTER_WIDTH)
        );
    headPointer_rd(READ_POINTER_WIDTH - WRITE_POINTER_WIDTH - 1
        downto 0) <= (others => '0');
    writePointer_rd(READ_POINTER_WIDTH - WRITE_POINTER_WIDTH - 1
        downto 0) <= (others => '0');

    cdc_readPointer: xpm_cdc_gray
        generic map (

            -- Common module generics
            DEST_SYNC_FF           => 2,
            INIT_SYNC_FF           => 1,
            REG_OUTPUT             => 0,
            SIM_ASSERT_CHK         => 0,
            SIM_LOSSLESS_GRAY_CHK  => 0,
            WIDTH                  => WRITE_POINTER_WIDTH
        )
        port map (

            src_clk      => rd_clk,
            src_in_bin   => readPointer(READ_POINTER_WIDTH-1 downto (
                READ_POINTER_WIDTH - WRITE_POINTER_WIDTH)),
            dest_clk     => wr_clk,
            dest_out_bin => readPointer_wr
        );

else generate

    cdc_headPointer: xpm_cdc_gray
        generic map (
            DEST_SYNC_FF           => 2,
            INIT_SYNC_FF           => 1,
            REG_OUTPUT             => 0,
            SIM_ASSERT_CHK         => 0,
            SIM_LOSSLESS_GRAY_CHK  => 0,
            WIDTH                  => READ_POINTER_WIDTH
        )
        port map (
            src_clk      => wr_clk,
```

```
            src_in_bin   => headPointer(WRITE_POINTER_WIDTH-1 downto
                (WRITE_POINTER_WIDTH - READ_POINTER_WIDTH)),
            dest_clk     => rd_clk,
            dest_out_bin => headPointer_rd
        );
    cdc_writePointer: xpm_cdc_gray
        generic map (
            DEST_SYNC_FF           => 2,
            INIT_SYNC_FF           => 1,
            REG_OUTPUT             => 0,
            SIM_ASSERT_CHK         => 0,
            SIM_LOSSLESS_GRAY_CHK  => 0,
            WIDTH                  => READ_POINTER_WIDTH
        )
        port map (
            src_clk      => wr_clk,
            src_in_bin   => writePointer(WRITE_POINTER_WIDTH-1 downto
                 (WRITE_POINTER_WIDTH - READ_POINTER_WIDTH)),
            dest_clk     => rd_clk,
            dest_out_bin => writePointer_rd
        );


    cdc_readPointer: xpm_cdc_gray
        generic map (

            -- Common module generics
            DEST_SYNC_FF           => 2,
            INIT_SYNC_FF           => 1,
            REG_OUTPUT             => 0,
            SIM_ASSERT_CHK         => 0,
            SIM_LOSSLESS_GRAY_CHK  => 0,
            WIDTH                  => READ_POINTER_WIDTH
        )
        port map (

            src_clk      => rd_clk,
            src_in_bin   => readPointer,
            dest_clk     => wr_clk,
            dest_out_bin => readPointer_wr(WRITE_POINTER_WIDTH-1
                downto (WRITE_POINTER_WIDTH - READ_POINTER_WIDTH))
        );
    readPointer_wr(WRITE_POINTER_WIDTH - READ_POINTER_WIDTH - 1
        downto 0) <= (others => '0');
end generate;

end architecture;
```

B The adjusted FELIX-ToHost decoding software

The new decoding function can be found in this appendix. The function is written in C++. The previous version can be found in the felix-star repository within the atlas-tdaq-felix group on the CERN GitLab server. However, essentially the only additions that were made were placed within conditional inclusion blocks that rely on CHUNK_HEADERS being defined.

```
int
decode_block(struct block_decoder* decoder, struct block* b)
{
  /* block address for unbuffered mode */
  decoder->block_address = ((uint64_t)b - decoder->dma_buffer_vaddr) & 0
      xFFFFFFFF;

  /* clear out any leftover chunks */
  if(decoder->state == block_decoder::DEC_STATE_SENDING) {
    LOG_TRACE("sending␣leftover␣data");
    int result;
    if(decoder->buffered_mode) {
      result = submit_chunk_buffered(decoder, decoder->scratch, decoder->
          scratch_pos);
    } else {
      result = submit_chunk_unbuffered(decoder, decoder->iov, decoder->
          iov_cnt);
    }
    if(0 == result) {
      decoder->state = block_decoder::DEC_STATE_NONE;
      decoder->iov_cnt = 0;
      decoder->scratch_pos = 0;
    } else {
      return BLOCK_AGAIN;
    }
  }

  unsigned trailer_size = decoder->trailer_size;

  unsigned n_subchunks = 0;
  unsigned expected_seqnr;
  unsigned start_subchunk = 0;

  if(decoder->reentry_n_subchunks == 0) {

#if REGMAP_VERSION < 0x0500
    unsigned block_size = decoder->block_size;
    uint16_t* raw_block = (uint16_t*)b;

    if (b->hdr.sob != 0xABCD){
      if(raw_block[1] == raw_block[0])
      {
        LOG_WARN("Block␣header␣%X␣%X␣with␣constant␣part␣overwritten.␣Not␣
            discarded␣(FLX-1829).",
                 raw_block[1], raw_block[0]);
      } else {
        LOG_ERR("Received␣invalid␣block␣header␣0x%x.␣Block␣discarded.", b
            ->hdr.sob, trailer_size);
```

```
        return BLOCK_ERROR;
      }
    }
#else
    unsigned block_size;

    if (b->hdr.sob == 0xABCD){ //only for tests! firmware rm-5 does not
       use 0xABCD
      block_size = 1024;
    } else {
      block_size = ((b->hdr.sob >> 8) - 0xC0 + 1) * 1024;
    }
    if (block_size != decoder->block_size) {
      LOG_ERR("received␣block␣header␣0x%x␣with␣irregular␣block␣size␣%d,␣
         expected␣%d.␣Block␣discarded.",
             b->hdr.sob, block_size, decoder->block_size);
      return BLOCK_ERROR;
    }
#endif

    expected_seqnr = (decoder->last_seqnr + 1) % 32;
    if(b->hdr.seqnr != expected_seqnr)
    {
      if(!(decoder->seqnr_err++ % 100)){
        LOG_DBG("received␣wrong␣sequence␣number:␣%d␣instead␣of␣%d␣(E-link
           :␣%d)",
            b->hdr.seqnr, expected_seqnr, b->hdr.elink);
      }
    }
    decoder->last_seqnr = b->hdr.seqnr;

// Skip the whole indexing subchunks part when using chunk headers.
#ifndef CHUNK_HEADERS
    // Go through the block to read all chunk trailers.
    unsigned pos = block_size - BLOCK_HEADER_SIZE;
    while(pos > 0) {
      subchunk_trailer_t* t = (subchunk_trailer_t*)(b->data + pos -
         trailer_size);

#if REGMAP_VERSION < 0x0500
      //OOB BUSY trailer (0xe05c)
      if(*(uint16_t*)t == 0xe05c){
        t = (subchunk_trailer_t*)(b->data + pos - trailer_size -
           trailer_size);
        pos-=2;
      }
#endif

      unsigned int sc_len = t->data.length;
      if (sc_len > block_size - BLOCK_HEADER_SIZE - trailer_size) {
        LOG_WARN("Block␣discarded␣due␣to␣inconsistent␣subchunk␣length␣%u"
           , sc_len);
        return BLOCK_ERROR;
      }

      unsigned int pad = PADDING(sc_len, trailer_size);
      pos -= sc_len + trailer_size + pad;
```

```
      if(pos > block_size - BLOCK_HEADER_SIZE) {
        LOG_WARN("Discarding truncated block. Pos %u pad %u sc_len %d
            subchunk lenght %d",
            pos, pad, t->data.length);
        return BLOCK_TRUNCATED;
      }

      decoder->subchunk_indices[n_subchunks].pos = pos;
      decoder->subchunk_indices[n_subchunks].trailer.value = t->value;
      n_subchunks++;
    }

#endif

  } else {
    LOG_TRACE("Reentry");
    n_subchunks = decoder->reentry_n_subchunks;
    start_subchunk = decoder->reentry_i;
    expected_seqnr = decoder->last_seqnr;
  }

#ifdef CHUNK_HEADERS
  // start_subchunk contains the index of the subchunk within the block (
      in bytes)
  // instead of the chunk number (in chunks) when using chunk headers.
  uint16_t pos = start_subchunk;

  while (pos < decoder->block_size - BLOCK_HEADER_SIZE) {
    subchunk_header_t header;

    // We know the headers are always aligned to 32 bits so the type
       punning here is allowed.
    header = *(subchunk_header_t*)(b->data + pos);

    decoder->error = header.data.trunc | (header.data.err << 2) | (header
        .data.crcerr << 3);

    int result = decoder->submit_subchunk(decoder, header.data.type, b->
        data + pos + sizeof(subchunk_header_t), header.data.length);
    if(result == BLOCK_AGAIN) {
      // In buffered mode, if decoder is in the DEC_STATE_SENDING
      // the scratch space already contains the current subchunk.
      decoder->reentry_i = pos;
      // For the if statement to recognize a reentry, n_subchunks is set
         to 1.
      decoder->reentry_n_subchunks = 1;
      return BLOCK_AGAIN;
    }

    pos += sizeof(header) + header.data.length + PADDING(header.data.
        length, sizeof(header));
  }

#else
  // walk subchunks in-order
  for(unsigned i=start_subchunk; i < n_subchunks; i++) {
```

```
    subchunk_t sc = decoder->subchunk_indices[n_subchunks - i - 1];
    uint32_t len = sc.trailer.data.length;
    uint32_t type = sc.trailer.data.type;
    decoder->error = sc.trailer.data.trunc // CHUNK_FW_TRUNC
          | (sc.trailer.data.err << 2) // CHUNK_FW_MALF
          | (sc.trailer.data.crcerr << 3); // CHUNK_FW_CRC

    int result = decoder->submit_subchunk(decoder, type, b->data + sc.pos
       , len);
    if(result == BLOCK_AGAIN)
    {
      // In buffered mode, if decoder is in the DEC_STATE_SENDING
      // the scratch space already contains the current subchunk.
      decoder->reentry_i = i + (decoder->buffered_mode && decoder->state
         == block_decoder::DEC_STATE_SENDING);
      decoder->reentry_n_subchunks = n_subchunks;
      return BLOCK_AGAIN;
    }

  }
#endif

  if(decoder->elink_entry->type == TTC){
    flush_ttc2h_buffer(decoder->elink_entry, decoder->block_address);
  }

  decoder->reentry_n_subchunks = 0;
  decoder->reentry_i = 0;
  return BLOCK_OK;
}
```