CERN Openlab project

Implementing the network debugging infrastructure for the new detector readout board

Christina Quast

October 12, 2015

CERN – LHCb

Supervisor: Niko Neufeld, Rainer Schwemmer

Contents

List of Figures

1 Introduction

For the next run of the LHCb experiment, new detectors are built, which use a different protocol from the one used for the old detectors in order to send the data collected from a collision. The data throughput will increase from the currently used 400 Gbit per second to astonishing 40 Tbit. The LHCb upgrade team built two FPGA boards: AMC40 and PCIe40 (see image 1.1). The AMC40 board has an Ethernet interface, produces UDP packets and sends them over the network. The PCIe40 board sends data over PCIe to a server, which in turn creates the network packets based on the PCIe data received, and sends them over the network. Each board is connected to the detectors and contains an FPGA, which does the preprocessing of the data received from the detector. The readout system is connected to the network and sends network packets to the backbone (see image 1.2). The readout system will operate at an overall speed of around 40 Tbit/s. In order to have a means of debugging all this network traffic, tools are necessary. The implementation of such a tool is the task for this Openlab Summer Student project.

A very commonly used tool when it comes to debugging network traffic is called Wireshark. It can display the network packet flow and dissect protocols, creating a neat view of the fields in the header of each layer of protocols. Furthermore, it allows for filtering of different protocol fields (e.g. the protocol http and a certain IP address of interest, see image 1.3). If there is an error in dissecting the packet, e.g. because the packet is corrupted and therefore does not conform to the standard form anymore, Wireshark will display an error message. Wireshark supports a large amount of protocols, but also provides a means for adding custom protocol dissectors by compiling them into the Wireshark code or adding them separately as plugins.

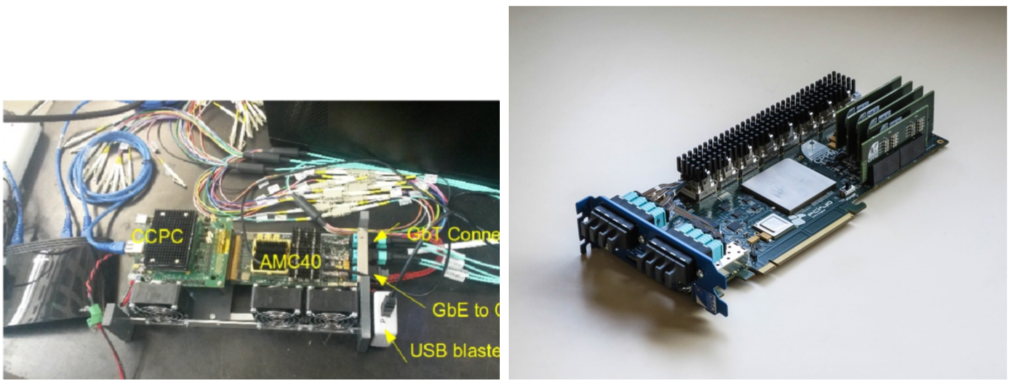

Figure 1.1: AMC40 board (left), PCIe40 board (right)

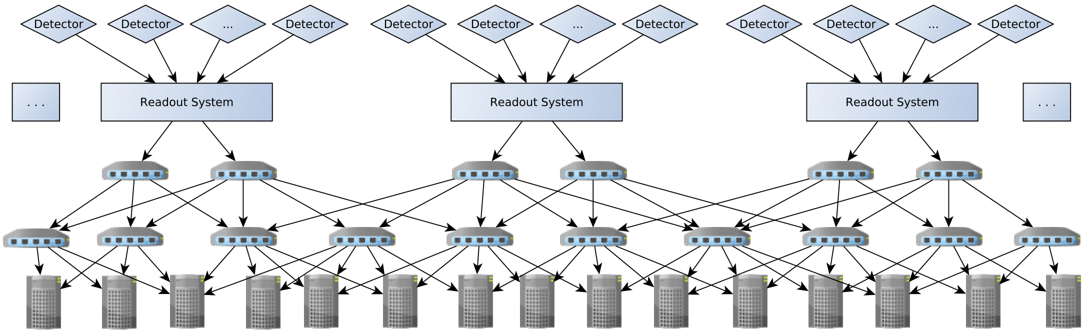

Figure 1.2: Readout board connection to detectors and network

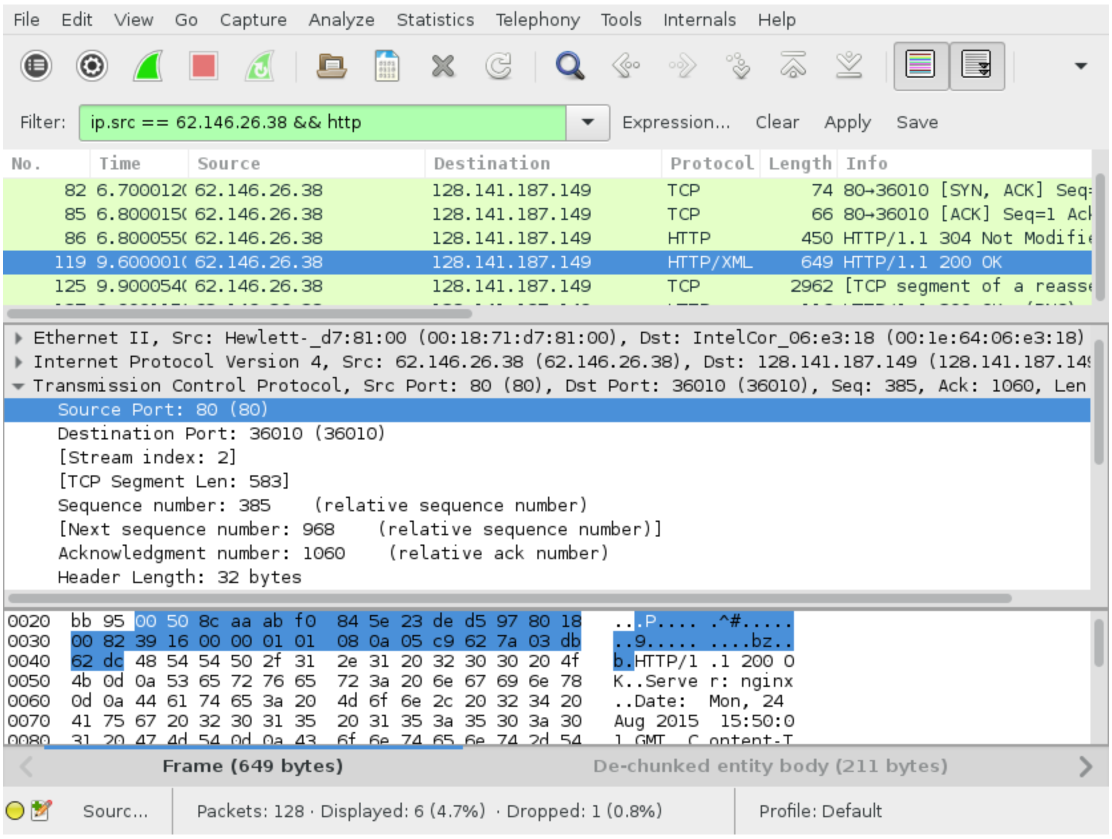

Figure 1.3: Wireshark with filtering for http traffic and IP source address

In case of the AMC40 board, data is sent as UDP packets, and can therefore directly be piped into Wireshark. In case of the PCIe40 board, the data is read out from the PCIe bus and delivered to userspace through a kernel driver, which is specifically created for this purpose. The kernel driver functions can be used through system calls. For sending the data to wireshark, a separate userspace program has to be written, which reads data from the kernel driver and wraps it into UDP packets before forwarding them to Wireshark.

1.1 Detector board to userspace interface

The interface functions to access kernel driver functions are listed in 1.1 and can be found in pcie40_driver/daq/daq.h in the lhcb−daq40−software git repository.

```
int p40_ctl_open(int dev);
void p40_ctl_close(int fd);

int p40_ctl_set_control(int fd, uint32_t ctl);
int p40_ctl_start(int fd);
int p40_ctl_stop(int fd);

uint32_t p40_ctl_get_status(int fd);

int  p40_daq_open(int dev, void **buffer);
void p40_daq_close(int fd, void *buffer);

int      p40_daq_set_read_off(int fd, uint32_t off);
uint32_t p40_daq_get_read_off(int fd);
uint32_t p40_daq_get_write_off(int fd);
uint64_t p40_daq_get_buf_size(int fd);
uint32_t p40_daq_get_msi_size(int fd);
uint64_t p40_daq_get_msi_nsecs(int fd);
```

Listing 1: Kernel driver interface

The function p40_meta_open returns a file descriptor and saves the pointer to the metadata in memory in one of the function arguments passed to it. The function call p40_daq_open works similar, but the pointer will point into the data area. The function p40_daq_set_read_off returns the offset, from which to read the current data element. p40_ctl_start starts data generation in the FPGA.

Only the metadata part of the kernel driver is implemented for now. The data part will follow in the next months.

1.2 The new readout protocol

The new detector protocol will consist of two packet sources. The metadata source contains all the header of the data and describes which type of data of which size to find at which offset from the beginning of the data. The data consists of consecutive areas of payload; length of each payload fragment is defined in the metadata.

The header used for the data generating test program amc40_capchecker can be found in amc40.hpp (see also listing 2).

```
struct __attribute__((__packed__)) amc40_hdr {
  uint32_t seqnum;
  uint16_t bytes;
  uint16_t frags;
  uint64_t evid;
};
```

Listing 2: MEP meta data structure

In each packet, what follows after the header are the fragments; each one of them with its own sub header, containing the sequence number (BXID) and payload length in nibbles or bytes, followed by data for each detector link (see image 1.4).

It is worth mentioning at this point, that the header structure explained is the header structure used for the readout systems currently under research. The structure will change in the future, in conjunction with the evolution of the readout system.

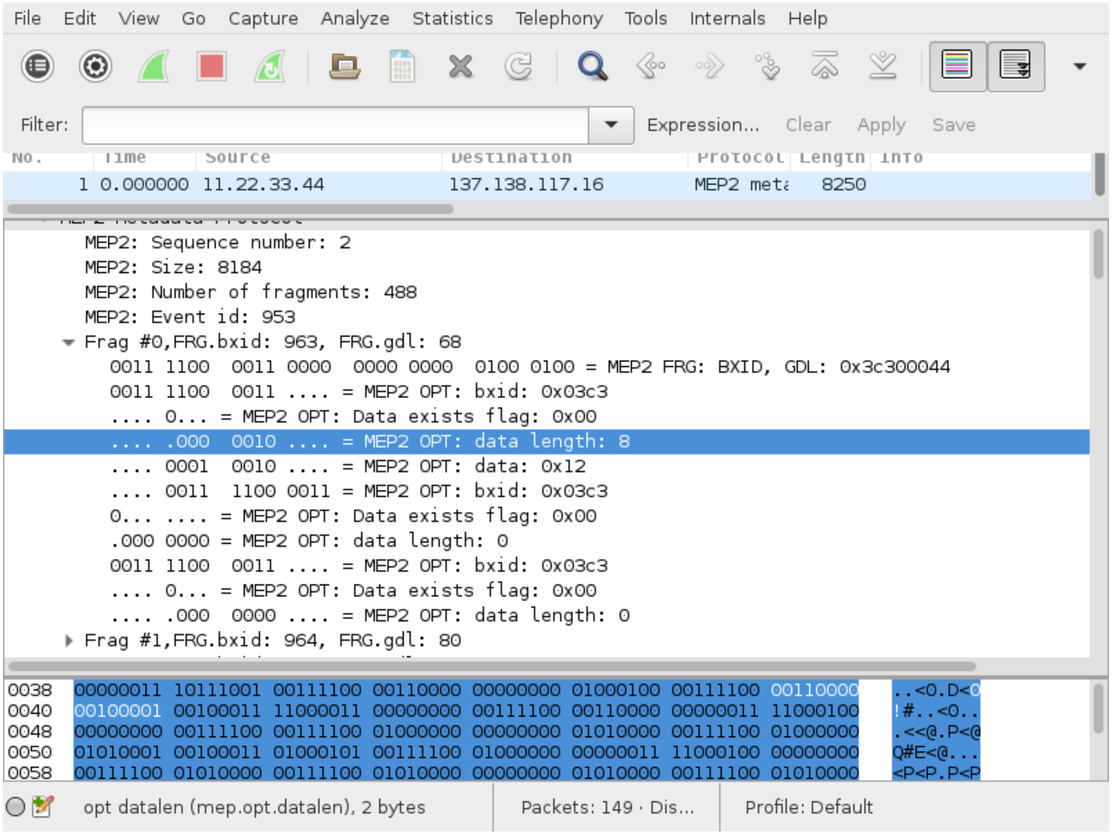

Figure 1.4: MEP protocol

2 Implementation

2.1 Wireshark basics

The wireshark manpage tells us: On Unix-compatible systems, the plugins are looked for in the following directories: the lib/wireshark/plugins/$VERSION directory under the main installation directory (for example, /usr/local/lib/wireshark/plugins/$VERSION), and then $HOME/.wireshark/plugins. It is possible to check whether a plugin was loaded by opening the About dialog box in Wireshark and looking at the Plugins tab.

2.2 Implementation of a wireshark dissector in C

Because of performance reasons, also there are Python for implementing a Wireshark dissector, the code will written in C. All the basic knowledge, how to implement the dissector, will be explained in the following subsection.

2.2.1 Basic dissector functions and structures

2.2.1.1 Example code for example protocol

In order to understand better, how wireshark dissectors are written, we take the protocol in image 2.1 as an example.

First, we need to register the protocol we want to use. The abbreviation can later be used as a filter string in wireshark.

```
void proto_register_mep(void)
{
    proto_mep = proto_register_protocol (
```

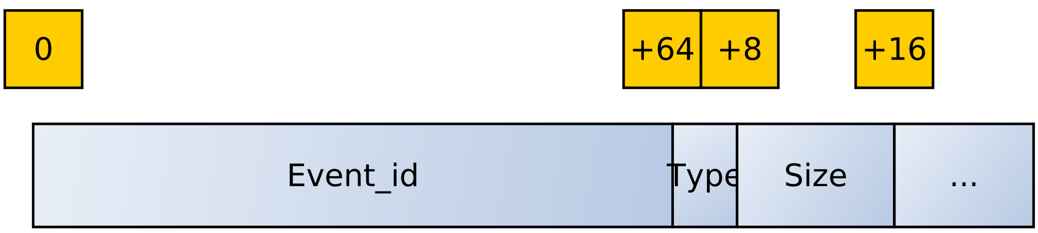

Figure 2.1: Example protocol structure

```
        "MEP Protocol", /* name       */
        "MEP",      /* short name */
        "mep"       /* abbrev     */
        );
}
```

The protocol dissector handle is registered, in our case in conjunction with the associated UDP port. In practice, probably any wireshark filter can be used here.

```
#define MEP_PORT    1234

void proto_reg_handoff_mep(void)
{
    static dissector_handle_t mep_handle;

    mep_handle = create_dissector_handle(dissect_mep, proto_mep);
    dissector_add_uint("udp.port", MEP_PORT, mep_handle);
}
```

For parsing packets, mainly the function proto_item∗ proto_tree_add_item(tree, id, tvb, start, leng is used. The parameter id describes how the field should be presented in wireshark. Some example values can be seen in the following source code.

```
void proto_register_mep(void) {
    static hf_register_info hf_data[] = {
        { &hf_data_evid,
            { "MEP2 data: Event ID", "mep.data.evid",
            FT_UINT64, BASE_DEC,
            NULL, 0x0,
            "evid", HFILL }
        },
        { &hf_data_type,
            { "MEP data: Type", "mep.data.type",
            FT_UINT8, BASE_DEC,
            VALS(data_type_names), 0x0,
            "data type", HFILL }
        },
        { &hf_data_size,
            { "MEP data: Size", "mep.data.size",
            FT_UINT16, BASE_DEC,
            NULL, 0x0,
            "data size", HFILL }
        },
    }
}
```

In order to parse the first three fields of the protocol, we would use the following source code.

```
static void dissect_mep(tvbuff_t *tvb, packet_info *pinfo,
    proto_tree *tree) {
    gint offset = 0;
    ...
    proto_tree *data_tree = proto_item_add_subtree(data_root,
            ett_data);
    proto_tree_add_item(data_tree, hf_data_evid, tvb, offset,
            8, ENC_BIG_ENDIAN);  // 64bit = 8 byte
    offset += 8;
    proto_tree_add_item(data_tree, hf_data_type, tvb, offset,
            1, ENC_BIG_ENDIAN);
    offset += 1;
    proto_tree_add_item(data_tree, hf_data_size, tvb, offset,
            2, ENC_BIG_ENDIAN);
    offset += 2;
    ...
    }
}
```

Without the wireshark plugin, the data can not be parsed by wireshark and the UDP payload looks like a binary blob (see 2.2). After copying the plugin into the respective wireshark plugin directory, the end result looks as seen in image 2.3.

2.2.2 Important wireshark dissector functions

The argument to the function tvb_get_bits* defined as tvbuff_t *tvb is the pointer to the buffer at that location, where the current protocol payload is stored. e.g., if UDP is the currently selected protocol, tvb will point to the first UDP payload element, just behind the UDP header (see code listing 2.2.2). From here, parsing of the data should start.

```
guint8 tvb_get_bits8(tvbuff_t *tvb, gint bit_offset, const gint no_of_bits);
guint16 tvb_get_bits16(tvbuff_t *tvb, guint bit_offset, const gint no_of_bits, const
guint32 tvb_get_bits32(tvbuff_t *tvb, guint bit_offset, const gint no_of_bits, const
guint64 tvb_get_bits64(tvbuff_t *tvb, guint bit_offset, const gint no_of_bits, const
```

Listing 3: Important dissector functions

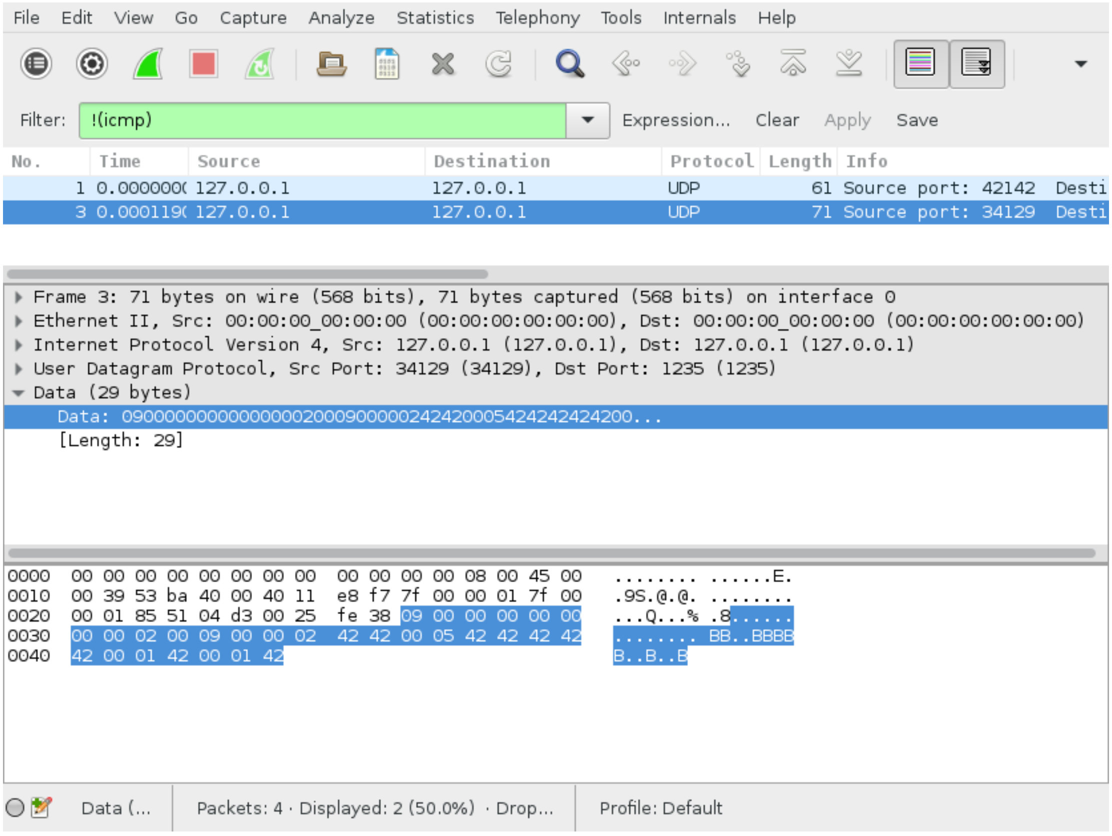

Figure 2.2: Wireshark without MEP dissector plugin

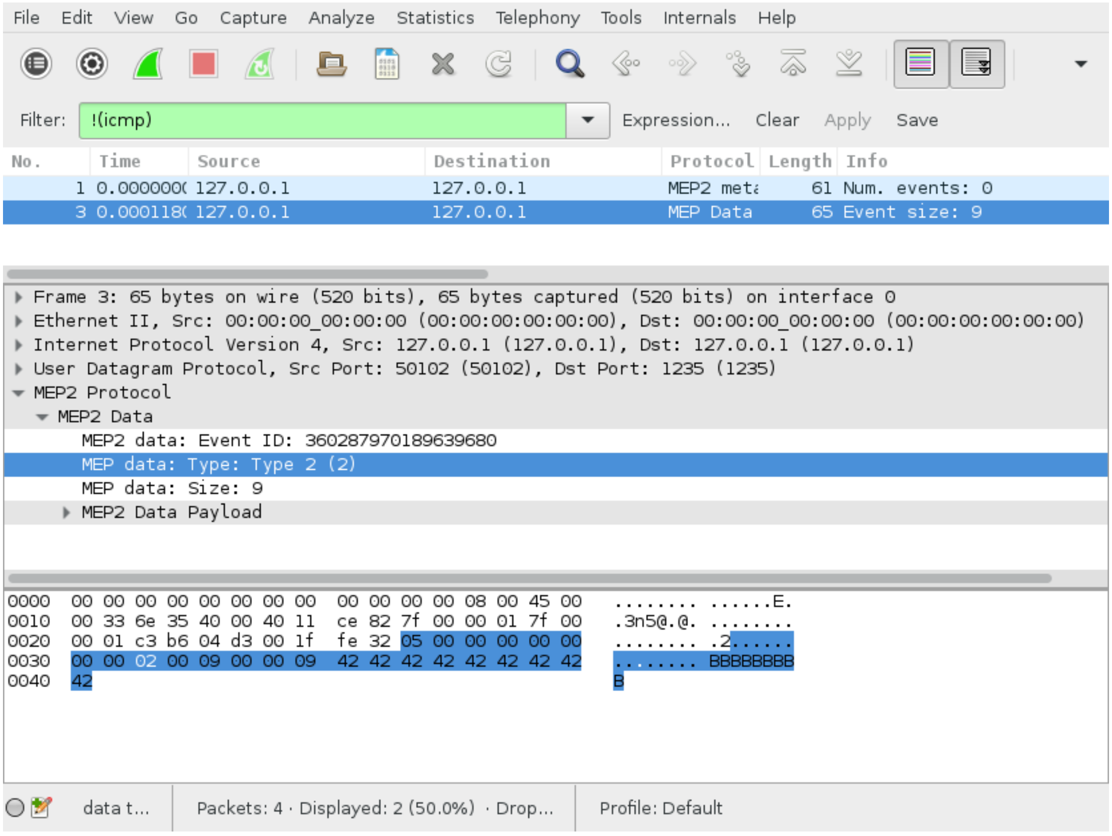

Figure 2.3: Wireshark with MEP dissector plugin

3 Conclusion

3.1 Accomplished tasks

A wireshark dissector was created, which can parse the MEP data provided by lhcb-daq40-software (see `https://git.cern.ch/web/lhcb-daq40-software.git`). The code for the dissector can be found on `https://github.com/chrysh/lhcb-daq40-dissector`. The plugin can be compiled using make. The plugin, which can then be found under .libs has then to be copied into the wireshark plugin directory before wireshark is started (see also section 2.1). For more and detailed information, refer to the wikipage `https://lbdokuwiki.cern.ch/doku.php?id=upgrade:logbook_openlab_2015`.

3.2 Known issues

There are still some tasks to be done. The config file, which is provided for each detector protocol used and defines the length of each field, is not read yet. Nevertheless, a header file is used to define the length of the fields for each protocol for now. The length is set in the initialization function of the dissector plugin, and can therefore be adjusted easily.

The code is not based on the data checker to be found under `https://git.cern.ch/web/lhcb-daq40-software.git`), because I needed to understand how wireshark dissectors work in the first place, and then how the lhcb−daq40−software code works. It would have been much better, if the code was actually based on the parser in lhcb−daq40−software in a modular way. In the current state, the dissector is written from scratch, with some inspiration taken from the amc40_capchecker code, which can be found in lhcb−daq40−software.

Furthermore, the special case of having a data payload field, which is larger than four bytes can not be displayed properly, while parsing the packet does not impose a problem. Calling the function proto_tree_add_bits_item with a larger value runs into an assertion error in proto.c (line 7604). To ensure correct parsing, a temporary solution is found, which is merely displaying the first 64 bit of the field and continuing parsing with the original data length value.

3.3 To be done in the future

After the functionality of the two boards, AMC40 and PCIe40, are ensured, a new network protocol has to be created, which will unite all the protocols for the different detector types. For that, another wireshark dissector has to be written. The dissector created for this project can be a good starting point for the next dissector implementation.

Reading the configuration file from a predefined directory has to be implemented in the future. Furthermore, a solution has to be found for the special case of having data which is larger than 64 bit.