FELIX Software Documentation

ATLAS FELIX Group

Version 818eb3f

Table of Contents

1. Introduction . . . 2
   1.1 Components . . . 2
2. FELIX STAR . . . 4
   2.1 Specifications . . . 4
      2.1.1 FELIX Card Interface . . . 5
      2.1.2 NetIO-next Interface . . . 5
      2.1.3 FELIX Bus Interface . . . 5
      2.1.4 FELIX Client Interface . . . 6
      2.1.5 Input and Output Formats . . . 6
      2.1.6 To Host Data Handling . . . 6
      2.1.7 To Flx Data Handling . . . 6
      2.1.8 Monitoring . . . 6
      2.1.9 Error Handling . . . 6
      2.1.10 BUSY Handling . . . 7
      2.1.11 TTC2Host Handling . . . 7
      2.1.12 E-Link setup . . . 7
      2.1.13 Handling of Stream IDs . . . 7
      2.1.14 Configuration . . . 7
   2.2 Implementation . . . 7
      2.2.1 Initialization . . . 7
      2.2.2 Data handling . . . 7
         2.2.2.1 To Host . . . 7
         2.2.2.2 To Flx . . . 8
      2.2.3 Monitoring . . . 8
   2.3 Performance . . . 9
   2.4 Issues and Improvements . . . 9
3. FELIX Client . . . 11
   3.1 Specifications . . . 11
4. NetIO next . . . 13
   4.1 Specifications . . . 13
      4.1.1 Communication Patterns . . . 13
      4.1.2 Communication Modes . . . 13
      4.1.3 Back-End System . . . 14
   4.2 Implementation . . . 14
      4.2.1 Architecture . . . 14
      4.2.2 High-Throughput and Low-Latency Sockets . . . 14
   4.3 Performance . . . 15
   4.4 Issues and Improvements . . . 17

5. FELIX Bus . . . 19
5.1 Specifications . . . 19
5.2 Implementation . . . 19
5.3 FELIX usage of FELIX Bus . . . 19
5.4 Performance . . . 20
5.5 Issues and Improvements . . . 20
6. FELIX Core . . . 22
6.1 Specifications . . . 22
6.1.1 FELIX Card Interface . . . 23
6.1.2 NetIO Interface . . . 23
6.1.3 FELIX Bus Interface . . . 23
6.1.4 Input and Output Formats . . . 23
6.1.5 To Host Data Handling . . . 24
6.1.6 From Host Data Handling . . . 24
6.1.7 Monitoring . . . 24
6.1.8 Error Handling . . . 24
6.1.9 BUSY Handling . . . 24
6.1.10 L1A Handling . . . 25
6.1.11 Filtering . . . 25
6.1.12 Handling of Stream IDs . . . 25
6.1.13 Configuration . . . 25
6.2 Implementation . . . 25
6.2.1 Initialization . . . 25
6.2.2 Data handling . . . 25
6.2.2.1 To Host . . . 25
6.2.2.2 From Host . . . 26
6.2.3 Monitoring . . . 26
6.3 Performance . . . 27
6.4 Issues and Improvements . . . 27
7. NetIO . . . 29
7.1 Specifications . . . 29
7.1.1 Communication Patterns . . . 29
7.1.2 Communication Modes . . . 29
7.1.3 Back-End System . . . 30
7.2 Implementation . . . 30
7.2.1 Architecture . . . 30
7.2.2 High-Throughput and Low-Latency Sockets . . . 30
7.3 Performance . . . 31
7.4 Issues and Improvements . . . 33
Appendix A. FELIX To Host Format . . . 35
A.1 Block Format . . . 35

A.1.1 Sub-chunk Format . . . 35
A.1.2 Length . . . 36
A.1.3 Chunk Error . . . 36
A.1.4 Truncation . . . 36
A.1.5 Sub-chunk Type . . . 36
A.1.6 Padding . . . 37
A.1.6.1 Enforce 16-bit alignment of sub-chunks . . . 37
Appendix B. FELIX From Host Format . . . 39
B.1 Formats . . . 39
Appendix C. FELIX Input Format . . . 41
C.1 Formats . . . 41
Appendix D. FELIX Output Format . . . 43
D.1 Formats . . . 43
References . . . 43

0 :!table: 0

1. Introduction

This document describes for each of the components in a running DAQ/DCS system:

- Specifications
- Implementation
- Performance
- Issues and Improvements

1.1 Components

2. FELIX Star	The main application(s) to communicate between FELIX Clients and FELIX Cards
3. FELIX Client	A generic client to connect over netio-next to FELIX Star using FELIX Bus
4. NetIO-next	A fast communication protocol for data messaging between FELIX Star and FELIX Clients
5. FELIX Bus	A communication bus for control messaging between FELIX Core/Star instances and their FELIX Clients
6. FELIX Core	The main application to communicate between FELIX Clients and FELIX Cards
7. NetIO	A fast communication protocol for data messaging between FELIX Core and FELIX Clients

Any notes in Italics are not yet implemented.

0 :!table: 1

2. FELIX STAR

2.1 Specifications

The FELIX Star application is the central process of a FELIX system, and replaces FELIX Core. Star stands for Single-Threaded-Asynchronous-Reactor. It tries to handle all its work in a single thread.

FELIX Star handles the communication between one or more FELIX Cards on one side and a set of NetIO-next clients on the other side. FELIX Star is a stateless system, to be run all the time, for DAQ and DCS purposes. It runs on a PC, running the Linux operating system, in which FELIX Cards and Network Cards are installed.

FELIX Star has the following functions:

1. Packet forwarding from the front-ends to the DAQ system: read and decode data packets from the FELIX Card and forward them as messages to NetIO-next clients, based on dynamic routing rules.
2. Packet forwarding from the DAQ system to the front-ends: receive messages from NetIO-next clients and forward them as packets to the FELIX Card E-Links as supplied in the message header.
3. Configure the FELIX Card, E-Link configuration and operational parameters based on the input of a configuration file. The FELIX Star system uses a number of FELIX Tools to handle this.
4. Recover from host failures. Using a publish/subscribe system host failures of FELIX clients can be handled transparently.
5. Advertise E-Link meta information to NetIO-next clients before they actually connect. This uses the global FELIX ID (FID) and the FELIX Bus.
6. Gather statistics and performance metrics and make those available in a virtual link in NetIO-next.
7. *Report operational status information, such as the status of the detector links, and warn in the case of hardware failure.*
8. Use the Global FELIX ID to address links in different cards serviced by different FELIX Star applications.
9. Handle the BNL-712 card, which internally is seen as two devices, by a multiple FELIX Star instances.
10. Handle multiple FELIX Cards by the multiple FELIX Star instance.

The FELIX Star Architecture

The figure shows a diagram of the architecture of the FELIX Star applications. In the case of a system with multiple FELIX Cards, multiple FELIX Star applications are run.

The following main FELIX Star applications are provided:

felix-tohost	Reads out data from a single device from a FELIX card and publishes its links on NetIO-next.
felix-toflx	Listens on NetIO-next and forwards data to a single device in a FELIX card.
felix-dir2bus	Reads internal info from felix-tohost and felix-toflx and publishes this on the FELIX Bus.
felix-fifo2elink	Reads a error or monitoring FIFO and publishes this on NetIO-next.

some more applications are provided for simulation and testing and can be found in the FELIX User Manual.

2.1.1 FELIX Card Interface

A FELIX Card can run or be programmed in GBT or in FULL Mode. For FELIX Star the handling of GBT or FULL Mode makes no difference other than that in GBT mode Links are made of GBT/E-Group/E-Path combinations and in FULL mode Links are just Links. In the following the word E-Link (GBT Mode) is used synonymously with FULL mode Link.

Per FELIX Card, and in the case of the BNL-712 per end-point on the card, 11 bits are reserved for the E-Link ID providing for a maximum of 2048 E-Links. A specification of a Global FELIX ID, which incorporates the E-Link, is available, to allow each subsystem to have their own range of E-Links.

Communication with the FELIX Card is done by using a driver. The driver gives access to:

- a register map of the card, for status and control, see Register Map[1]
- two DMA channels for fast data transfer in and out of the card, one DMA channel in each direction
- *more DMA channels are being implemented, which would enlarge the throughput of the system*
- interrupts to react to the cards needs

To make it easier to interface with the three items above the FLX Card API was created.

2.1.2 NetIO-next Interface

Communication with NetIO-next is done via the NetIO-next Library which supplies:

- a publish-subscribe model, allowing clients to subscribe to certain data on a node/port combination
- buffered and unbuffered (zero-copy) data communication

2.1.3 FELIX Bus Interface

At the startup of FELIX Star a list of E-Links and host/port combinations will be published in a set of tables on FELIX Bus. It does so by writing the information into a file which is read by felix-dir2bus and which publishes it on the FELIX Bus. Any FELIX Bus client, including any NetIO-next client, finds E-Links by subscribing to the FELIX Bus and looking up the E-Link in the table to find out where to connect.

2.1.4 FELIX Client Interface

To facilitate an easy send and subscribe model, a simple client interface was designed, such that a client of the FELIX system would not need to know about FELIX Bus or NetIO-next. The FELIX Client has a few methods for sending data to a FELIX card and subscribing to data from a FELIX card, all using the global FELIX IDs.

2.1.5 Input and Output Formats

Input into and Output from FELIX Star, both on the FELIX Card and NetIO-next side, is documented in the following formats:

Name	Type	Description
FELIX To Host	DAQ, DCS/SCA	Reading from the FELIX Card
FELIX From Host	DCS/SCA	Writing to the FELIX Card
FELIX Input	DCS/SCA	Input from FELIX Clients
FELIX Output	DAQ, DCS/SCA	Output to FELIX Clients

2.1.6 To Host Data Handling

Data is read from the card, via DMA into the memory of the PC, and then routed according to E-Link to NetIO-next subscribers. ToHost data (chunks) are split, by the FELIX firmware on the card, into sub-chunks to fit in 1024 Byte blocks with an E-Link and Sequence number in the header. *Future firmware allows for larger block sizes.* FELIX Star reads blocks, keeps them by E-Link, checks Sequence numbers, and recombines sub-chunks into chunks of data. These chunks of data are subsequently published via NetIO-next, prepended with the FELIX header. Blocks from the card in the ToHost direction can contain any E-Link. Blocks of one E-Link must arrive in order with Sequence number incremented by one.

2.1.7 To Flx Data Handling

NetIO-next Clients may send data to FELIX Star over NetIO-next, which is then copied to the card using DMA or single shot. Upon arrival of a message in NetIO-next the header is inspected for the E-Link and data is sent to the respective E-Link. Data sent to FELIX Star has to be encoded as 32 byte packets with a 2 byte header, which specifies the E-Link.

2.1.8 Monitoring

Monitoring information is made available in JSON format[2] on a FIFO. The felix-fifo2elink application is able to publish this FIFO into NetIO-next where FELIX clients can subscribe to it. *A web application to show statistics is foreseen. An application to connect this E-link to ATLAS ERS is also being prepared*

2.1.9 Error Handling

Errors in the data stream are signaled as part of the sub-chunks, see FELIX To Host Format. Errors, such as CRC and 8B/10B encoding errors, are forwarded into the status field of the FELIX Header

and handled in-stream, see FELIX Output Format. Errors in the FELIX Star applications are routed to a FIFO in JSON format. The felix-fifo2elink application will publish then to NetIO-next, where any FELIX client can subscribe to them.

2.1.10 BUSY Handling

To identify who raised the BUSY on the Card, FELIX Star provides a virtual BUSY Info E-Link which shows the origin of the BUSY signal. In the case of internal buffers running full FELIX star could possibly provide a list of E-Links.

2.1.11 TTC2Host Handling

L1A information coming in via the TTC cable on the FELIX card can be routed to any E-Link. FELIX Star merely routes the information to any NetIO-next subscribers. FELIX Star configures the E-Link number and its width.

2.1.12 E-Link setup

In the FELIX Tool set a separate application called 'elinkconfig'[3] or feconf can be used to handle settings of the FELIX Card, such as E-Link configuration.

2.1.13 Handling of Stream IDs

The handling of StreamIDs is part of the globa FELIX ID. The firmware and FELIX Star can be setup to handle StreamIDs.

2.1.14 Configuration

All E-Links will be configurable by feconf. Configuration will happen when FELIX Star is started, by using feconf.

2.2 Implementation

2.2.1 Initialization

The initialization of the FELIX Cards is done by a the FELIX Tools flx-init and feconf.

2.2.2 Data handling

In FELIX Star the handling of To Host and To Flx is entirely separated and described below.

2.2.2.1 To Host

TBC

The DMA of the FELIX Card is setup in continuous mode, filling a contiguous buffer of memory, allocated by the cmem_rcc driver[4]. A single thread is responsible for reading data from the contiguous memory. Data is encoded in 1024 Byte blocks. The header of a block is inspected for its E-Link and sequence number. The sequence number is verified not to be out of order. Track is kept

in a table of E-Links and current sequence numbers. Based on the E-Link a block is copied to one of multiple block queues. The same E-Link always goes to the same block queue. For each block queue a worker thread handles the processing of the blocks. Within the worker thread the blocks will run through a pipeline using the following steps:

1. The block is copied from contiguous memory into a buffer. This buffer is pre-allocated in the worker thread and retrieved from form a queue of free blocks. At this point the memory in the contiguous buffer can be made available for further DMA transfers by advancing the read pointer.
2. The next step is to decode the block into variable-length packets also known as chunks. Sub-chunks of in previous blocks are prepended to the current sub-chunk to create a data chunk, if possible, or the block is kept to wait for more data to come in.
3. The next action is to gather statistics. This step counts sub-chunks and some other entities processing it into some central data structure, which uses atomic variables to ensure thread synchronization. Histograms are also built using special queues to make sure they have no impact on overall system performance. All this information including configuration of FELIX Core itself is made available to FELIX Monitoring.
4. In this step information is extracted from the chunks. This information is used to route the data packets in the next step. At this time only the E-Link is stored. In future this could be extended to extract an event identifier or other information.
5. The meta information from the previous step is used to assign one or more NetIO clients for the chunk.
6. In this final step the chunk is sent to the NetIO clients assigned in the previous step.

Copies of data, which happen to be expensive, happen in the first and last step of the process. The first step is needed to free up, as fast as possible, the contiguous buffer of memory the DMA writes into. In the last step a copy is needed to transfer data to the underlying network stack. The second copy can be avoided by using NetIO's RDMA capabilities.

For testing purposes the FELIX Card reader can be replaced at runtime by a file reader, which reads data from a file or a data generator, which generates random data. In both cases FELIX Core will see this data as if it was read from a FELIX Card.

2.2.2.2 To Flx

Traffic that is sent from NetIO Clients to the frontend is handled by a single thread. Messages from NetIO Clients need to be encoded as 32 bytes including a 2 byte header. The header is decoded and the data is routed to the respective E-Link. HDLC encoding of data is to be done on the NetIO client side with a special library, which is a deliverable of the FELIX project and is available. FELIX Core is not aware of the fact that an E-Link is encoded with HDLC. FELIX Core monitors the number of blocks that are sent to each E-Link. This information is available via the FELIX monitoring tools and the FELIX Core web frontend.

2.2.3 Monitoring

Statistics and configuration information is made available in JSON format by FELIX Core. Internally it runs a small web-server which supplies this information on request. For testing purposes FELIX

Core provides a few web pages displaying this information, but as the web server is fairly simple, one should restrict the number of web clients to one or two.

For extensive monitoring of FELIX hosts we have multiple tools:

1. The tool 'felix-mon' to log FELIX data metrics to disk
2. A standalone web application can display current and historical monitoring metrics of FELIX hosts in the same network
3. An integration of FELIX with the TDAQ PBEAST database, that essentially connects FELIX to the TDAQ monitoring infrastructure.

2.3 Performance

TBD - Full-chain measurements ongoing.

2.4 Issues and Improvements

The following issues and possible improvements are known:

1. Busy handling needs to be implemented.
2. Error handling needs to be implemented.
3. Filtering needs to be specified and implemented.
4. E-Link StreamIDs need to be specified and implemented.
5. FELIX Core does not handle configuration, 'elinkconfig' is currently used to do so. We may want to take code from 'elinkconfig' and create a library for configuration. This library can then be used by 'elinkconfig' and FELIX Core for doing the configuration.
6. Lockout may occur if in the FromHost direction large data is sent (split in 32 byte packets) to a single E-Link. As there is currently just one thread handling this direction, we may be spending too much time on large data. We would better use some round-robin algorithm to handle queues from different NetIO subscribers, not to lockout anyone.

0 :!table: 2

3. FELIX Client

3.1 Specifications

:!table: 3

4. NetIO next

4.1 Specifications

4.1.1 Communication Patterns

TBR

The first communication pattern in NetIO is simple point-to-point communication. In this pattern a sender sends a series of messages to a receiver. The established connection is reliable and preserves order of messages. NetIO sockets are used to establish connections and send messages. A send socket can connect to exactly one receive socket. A receive socket can receive messages from multiple send sockets. The send/receive pattern is illustrated in the following figure.

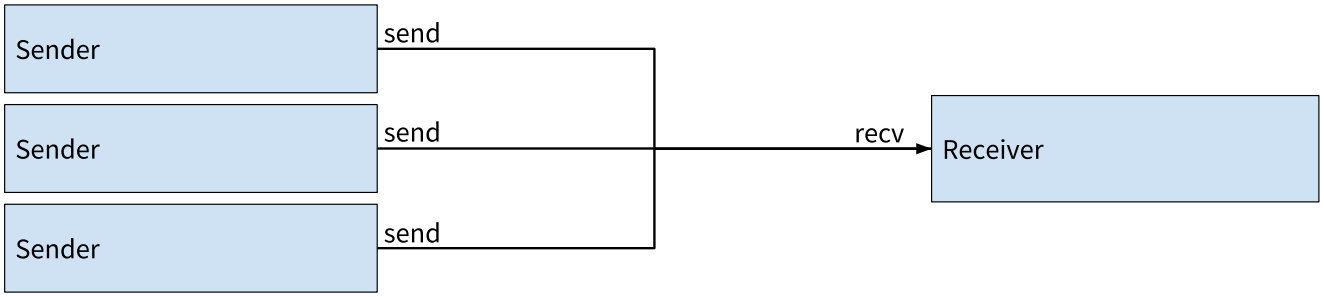


Figure 1. Send/Receive Pattern

The second communication pattern is publish/subscribe communication. A publish/subscribe system, as illustrated in the Figure below, has two types of actors: subscribers, which receive data, and publishers, which publish data. The publisher does not have any prior knowledge of the subscribers. Instead, subscribers send a subscription request to the publishers. Publishers receive the subscription request and add the subscriber to a subscription table. When publishing a message, a publisher looks up the subscription in the subscription table and sends the message to all subscribers. A subscriber can specify in the subscription request a mask to filter the messages that it will receive.

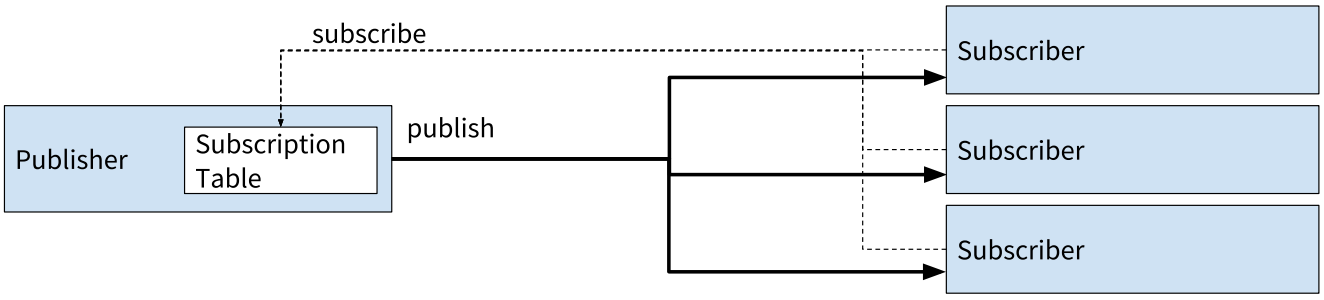


Figure 2. Publish/Subscribe Pattern

4.1.2 Communication Modes

NetIO distinguishes between two principal communication modes: buffered high-throughput communication and unbuffered low-latency communication. The relevant send and receive sockets operate differently in several aspects including the use of threads, use of the event loop, use of callbacks and use of buffers.

4.1.3 Back-End System

NetIO has a back-end system to support different network technologies and APIs. Currently two different back-ends exist. The first back-end uses POSIX sockets to establish reliable connections between endpoints. Primarily this back-end is used on TCP/IP connections in Ethernet networks, but in principle a technology like SDP could also be used. The second back-end is based on libfabric. It is used for communication via Infiniband as well as RDMA-over-Ethernet technologies.

4.2 Implementation

4.2.1 Architecture

The NetIO architecture is illustrated in the following Figure.

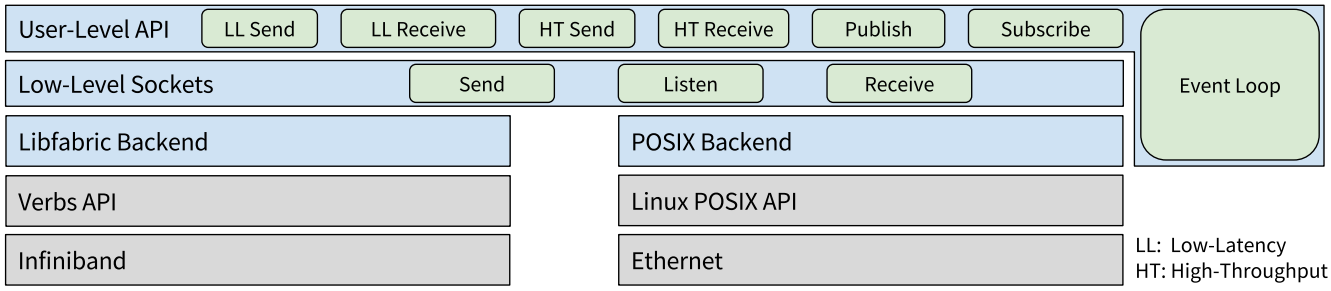


Figure 3. NetIO Architecture

There are two software layers within NetIO. The upper level contains user-level sockets. These are the sockets that application code interacts with. The different socket types are listed in Table~\ref{table:sockets}.

The lower architecture level provides a common interface to the underlying network APIs. The common interface consists of three low-level socket types (a send socket, a listen socket and a receive socket), which are implemented by each back-end. The low-level sockets provide basic connection handling and simple transmission of messages between two endpoints. All higher level functionality like buffering, notification of user code via callbacks, or the publish/subscribe system are implemented in the user-level sockets. This maximizes code sharing among the back-ends, as only code that is specific to the underlying network technology is implemented in the low-level sockets.

Both architecture levels use a central event loop to handle I/O events like connection requests, transmission completions, error conditions, or timeouts. The event loop is executed in a separate thread. Its implementation is based on the epoll framework in the Linux kernel.

Network endpoints are addressed by IP address and port, even for back-ends that do not natively support this form of addressing. For the Infiniband back-end the librdma compatibility layer is used to enable addressing by IP and port.

4.2.2 High-Throughput and Low-Latency Sockets

High-throughout sockets and low-latency sockets work fundamentally different. The differences are described in this section.

A high-throughput send socket does not send out messages immediately but maintains buffers in which messages are copied. The buffers are referred to as pages. Due to the buffering fewer, larger packets are sent on the network link. This approach is more efficient and yields a higher throughput. However, the average transmission latency of any specific message is increased due to the buffering. Once a page is filled the whole page is sent to the receiving end. Additionally, a timer (driven by the central event loop) flushes the page at regular intervals to avoid starvation and infinite latencies on connections with a low message rate. A typical timeout interval is 2\,s. A message is split if it does not fit into a single page. The original message is reconstructed on the receiving side.

A high-throughput receive socket receives pages that contain one or more messages or partial messages. The messages are encoded by simply prepending an 8\,byte length field to the message contents. The high-throughput receive socket maintains two queues: a page queue, which contains unprocessed pages that have been received from a remote, and a message queue, in which messages are stored that are extracted from pages when they are processed. The high-throughput receive socket enqueues received pages in the receive page queue. When user code calls \texttt{recv()} on a high-throughput receive socket, it will return the next message from the message queue. If the message queue is empty, the next page from the receive page queue is processed and the contained messages are stored in the message queue. When processing a received page the contained messages are copied. A new zero-copy mode is currently under development in which it is also possible to avoid this additional copy.

A low-latency send socket does not buffer messages. Messages are immediately sent to the remote process. Unlike for high-throughput send sockets, there is also no additional copy: the message buffer is directly passed to the underlying low-level socket. These design decisions minimize the added latency of a message send operation.

A low-latency receive socket handles incoming messages by passing them to the application code via a user-provided callback routine, instead of enqueuing the messages in a message queue. This approach enables incoming messages to be processed immediately. The message buffer is only valid during execution of the callback. After execution of the callback routine the receive buffer will be freed. If necessary, a user can decide to copy the buffer in the callback routine.

4.3 Performance

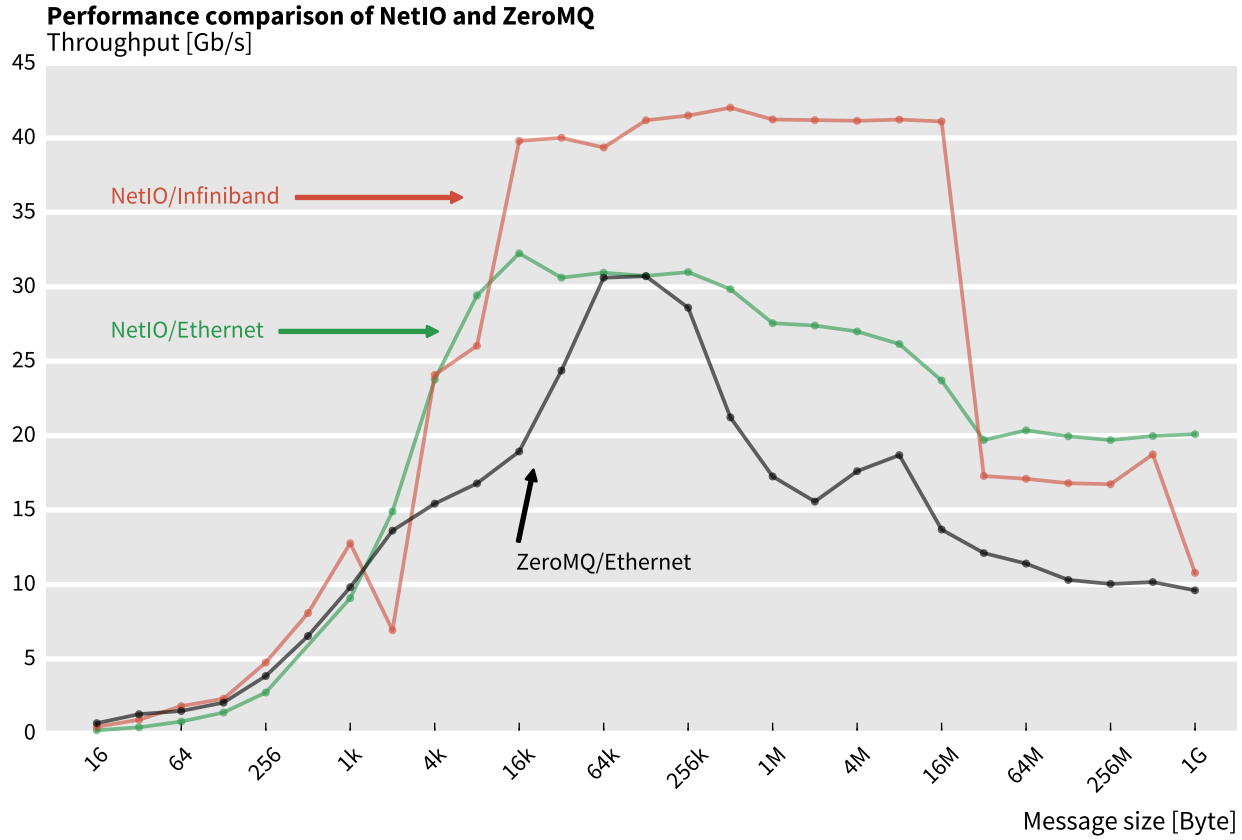


Figure 4. NetIO Throughput

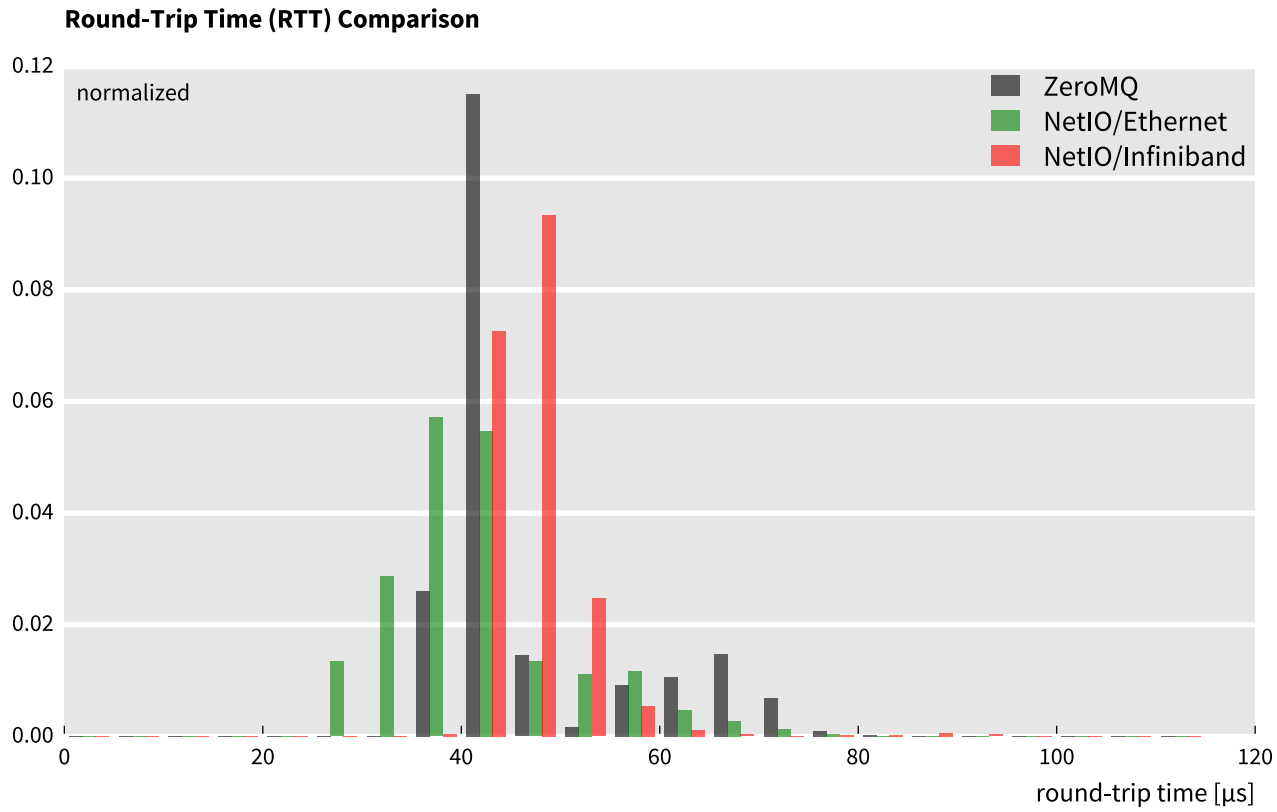


Figure 5. NetIO Round Trip Time

4.4 Issues and Improvements

0 :!table: 4

5. FELIX Bus

5.1 Specifications

FELIX Bus supports communication between different instances of FELIXCore and their clients, such as the SWROD[5].

FELIX Bus has the following features:

- FELIX Bus can have any number of clients. Clients can be publishers of information, subscribers of information or both.
- Information is published on the FELIX Bus, which is distributed to all subscribers on the FELIX Bus.
- When a publisher updates information on FELIX Bus, all subscribers are notified.
- Subscribers can do lookups of information at any time.
- When a subscriber unsubscribes from FELIX Bus, intentionally or due to failure, the information published by this subscriber is retracted from FELIX Bus and other subscribers are informed.
- Within Felix the FELIX Bus is used to distribute tables with E-Link->host/port association as well as tables for monitoring purposes.
- There is no single point of failure within FELIX Bus, it exists with its information as long as there is at least one client.
- FELIX Bus supports auto discovery of clients within a network.

5.2 Implementation

FELIX Bus is a virtual bus. Any client just needs to link with the FELIX Bus library, and publish information and or subscribe to information. FELIX Bus is a thin layer on top of ZeroMQ[6] and Zyre[7].

FELIX Bus announces its client via the Zyre beacon system, using a network broadcast. Internally this uses the UDP protocol. Each client is notified of the arrival or departure of any other client, within the network.

Once a client is known, further communication takes place via ZeroMQ, using the TCP/IP protocol. The ZeroMQ publish/subscribe mechanism is used to distribute information between different clients. A new client requests each other client for its information, thus building the full set of information. Each client replies on these requests.

5.3 FELIX usage of FELIX Bus

In FELIX the FELIX Bus is used for distribution and synchronization of two maps:

- ElinkTable, which links **E-Link** to **FelixId** and **peerId**

- FelixTable, which links **FelixId** to **host/port** and **peerId**

in which:

- **elink** is a 11 bit E-Link or link ID used inside Felix.
- **felixId** is a unique id or address per FELIX Core instance. *This needs to be sorted out still with regards to the Global FELIX ID proposal*
- **host/port** is the address where a Felix client needs to connect to.
- **peerId** is an internal ZeroMQ/Zyre ID to use when a client fails or disappears.

These two FELIX tables allow lookup of E-Link->host/port information.

FELIX Bus handles the failure of a client by removing the entries with a particular peerId from any of the tables used.

5.4 Performance

No performance measurements have been done.

5.5 Issues and Improvements

The following improvements may be implemented:

- Proper API to deal with both the publishing of E-Links and addresses as well as their lookups, without having to deal with the two tables, which are used internally in FELIX Bus.
- More information about the E-Links (e.g. width) may be published.
- The actual list(s) may be too large to fully distribute to all clients. To avoid this one could foresee 2 or 3 servers, of which the hostnames are distributed via FELIX Bus. Theses servers would contain the full tables, which are partially handed to them via the publishing clients. Each of the publishing clients would thus only distribute their partial tables to 2 or 3 servers. Subscribing clients would be notified when the tables change, and would do single or multiple lookups of limited number of E-Links.

0 :!table: 5

6. FELIX Core

6.1 Specifications

The FELIX Core application is the central process of a FELIX system. It handles the communication between one or more FELIX Cards on one side and a set of NetIO clients on the other side. FELIX Core is a stateless system, to be run all the time, for DAQ and DCS purposes. It runs on a PC, running the Linux operating system, in which the FELIX Card(s) and Network Card(s) are installed.

FELIX Core has the following functions:

1. Packet forwarding from the front-ends to the DAQ system: read and decode data packets from the FELIX Card and forward them as messages to NetIO clients, based on dynamic routing rules.
2. Packet forwarding from the DAQ system to the front-ends: receive messages from NetIO clients and forward them as packets to the FELIX Card E-Links as supplied in the message header.
3. Configure the FELIX Card, E-Link configuration and operational parameters based on the input of a configuration file.
4. Recover from host failures. Using a publish/subscribe system host failures of FELIX clients can be handled transparently.
5. Advertise E-Link meta information to NetIO clients before they actually connect. *Use the Global FELIX ID.*
6. Gather statistics and performance metrics and make those available.
7. Report operational status information, such as the status of the detector links, and warn in the case of hardware failure.
8. Offset E-Links to distinguish when multiple FELIX Core applications are run to read out multiple cards. *Use the Global FELIX ID.*
9. Handle the BNL-711 card, which internally is seen as two cards, by a single FELIX Core instance.
10. *Handle multiple FELIX Cards (including BNL-711) by the same FELIX Core instance.*

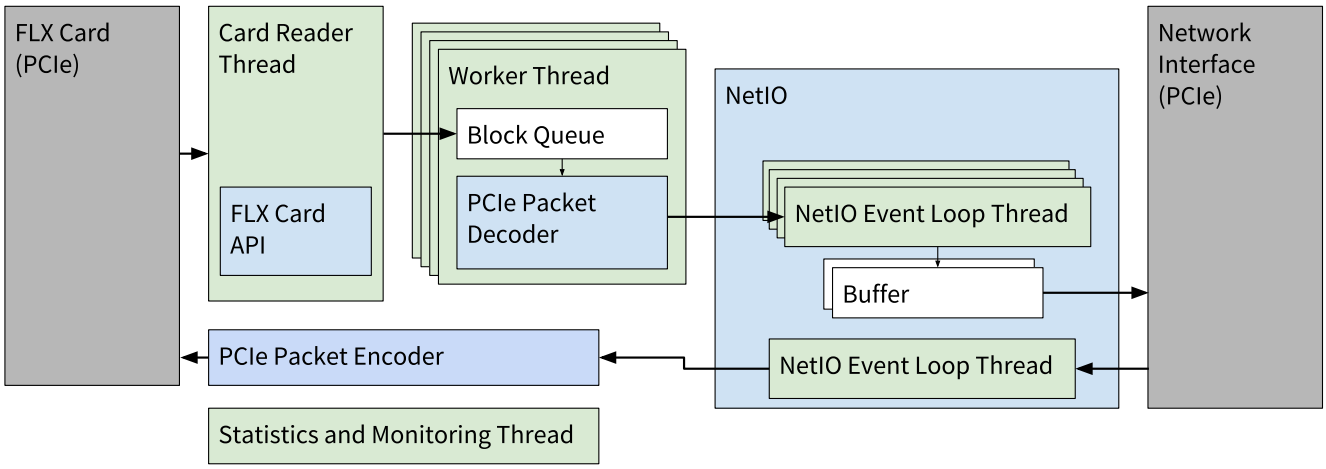


Figure 6. The FELIX Core Architecture

The figure shows a diagram of the architecture of the FELIX Core application. In the case of a

system with multiple FELIX Cards, multiple FELIX Core applications are run.

6.1.1 FELIX Card Interface

A FELIX Card can run or be programmed in GBT or in FULL Mode. For FELIX Core the handling of GBT or FULL Mode makes no difference other than that in GBT mode Links are made of GBT/E-Group/E-Path combinations and in FULL mode Links are just Links. In the following the word E-Link (GBT Mode) is used synonymously with FULL mode link.

Per FELIX Card, and in the case of the BNL-711 per end-point on the card, 11 bits are reserved for the E-Link ID providing for a maximum of 2048 E-Links. *A specification of a Global FELIX ID, which incorporates the E-Link, is being prepared, to allow each subsystem to have their own range of E-Links.*

Communication with the FELIX Card is done by using a driver. The driver gives access to:

- a register map of the card, for status and control, see Register Map[1]
- two DMA channel for fast data transfer in and out of the card, one DMA channel each
- interrupts to react to the cards needs

To make it easier to interface with the three items above the FLX Card API was created.

6.1.2 NetIO Interface

Communication with NetIO is done via the NetIO Library which supplies:

- a publish-subscribe model, allowing clients to subscribe to certain data on a node/port combination
- low-latency and high-throughput data communication

6.1.3 FELIX Bus Interface

At the startup of FELIX Core a list of E-Links and host/port combinations will be published in a set of tables on FELIX Bus. Any FELIX Bus client, including any NetIO client, finds E-Links by subscribing to the FELIX Bus and looking up the E-Link in the table to find out where to connect.

6.1.4 Input and Output Formats

Input into and Output from FELIX Core, both on the FELIX Card and NetIO side, is documented in the following formats:

Name	Type	Description
FELIX To Host	DAQ, DCS/SCA	Reading from the FELIX Card
FELIX From Host	DCS/SCA	Writing to the FELIX Card
FELIX Input	DCS/SCA	Input from FELIX Clients
FELIX Output	DAQ, DCS/SCA	Output to FELIX Clients

6.1.5 To Host Data Handling

Data is read from the card, via DMA into the memory of the PC, and then routed according to E-Link to NetIO subscribers. ToHost data (chunks) are split, by the FELIX firmware, into sub-chunks to fit in 1024 Byte blocks with an E-Link and Sequence number in the header. FELIX Core reads blocks, sorts them by E-Link, checks Sequence numbers, and recombines sub-chunks into chunks of data. These chunks of data are subsequently published via NetIO, prepended with the FELIX header. Blocks in the ToHost direction can contain any E-Link, meaning there is no order of E-Links, but blocks of one E-Link must arrive in order with Sequence number incremented by one.

6.1.6 From Host Data Handling

NetIO Clients may send data to FELIX Core over NetIO, which is then copied to the card using DMA. Upon arrival of a message in NetIO the header is inspected for the E-Link and data is sent to the respective E-Link. Data sent to FELIX Core has to be encoded as 32 byte packets with a 2 byte header, which specifies the E-Link.

6.1.7 Monitoring

Monitoring information is made available in JSON format[2] using an internal web server. A set of simple web pages can display this information by connecting to a FELIX Core instance. Programs such as 'felix-mon' connect to multiple FELIX Core instances, provide history and forward monitoring information to the ATLAS DAQ system.

6.1.8 Error Handling

Errors in the data stream are signaled as part of the sub-chunks, see FELIX To Host Format. Errors, such as CRC and 8B/10B encoding errors, are forwarded into the status field of the FELIX Header, see FELIX Output Format. The same information is also published on a virtual ERROR E-Link.

6.1.9 BUSY Handling

The handling of the BUSY both in Firmware and Software is under discussion. _A BUSY can be raised by software by setting the correct register in the FELIX Card. This is combined with the BUSY of the card itself and output on the BUSY Lemo connector.

FELIX Core raises the BUSY if one of the following conditions is met:

1. A BUSY signal was raised by at least one of the NetIO clients (e.g. SWROD). This signal is received via a separate virtual BUSY E-Link.
2. The internal output buffers towards NetIO are getting full. In this case enough space should be available after raising the BUSY to handle all incoming data until the front-ends stop sending data.

To identify who raised the BUSY on the Card, FELIX Core provides a virtual BUSY Info E-Link which shows the origin of the BUSY signal. In the case of internal buffers running full FELIX Core could possibly provide a list of E-Links._

6.1.10 L1A Handling

L1A information coming in via the TTC cable on the FELIX card can be routed to any E-Link. FELIX Core merely routes the information to any NetIO subscribers. *FELIX Core configures the E-Link number and its width.*

In the FELIX Tool set a separate application called 'elinkconfig'[3] can be used to handle settings of the FELIX Card, such as E-Link configuration.

Currently 'elinkconfig' handles all configuration.

6.1.11 Filtering

Filtering of ToHost traffic is foreseen, but has not been specified nor implemented.

6.1.12 Handling of Stream IDs

The handling of StreamIDs both in Firmware and Software is under discussion.

6.1.13 Configuration

All E-Links will be configurable by FELIX Core. A FELIX ID scheme and the configuration of E-Links is under discussion. Configuration will happen when the FELIX Core is started.

Currently 'elinkconfig' handles all configuration.

6.2 Implementation

6.2.1 Initialization

The initialization of the FELIX Cards is done by FELIX Core. **At this time any real initialization of links is done using 'elinkconfig'.**

6.2.2 Data handling

In FELIX Core the handling of To Host and From Host is entirely separated and described below.

6.2.2.1 To Host

The DMA of the FELIX Card is setup in continuous mode, filling a contiguous buffer of memory, allocated by the cmem_rcc driver[4]. A single thread is responsible for reading data from the contiguous memory. Data is encoded in 1024 Byte blocks. The header of a block is inspected for its E-Link and sequence number. The sequence number is verified not to be out of order. Track is kept in a table of E-Links and current sequence numbers. Based on the E-Link a block is copied to one of multiple block queues. The same E-Link always goes to the same block queue. For each block queue a worker thread handles the processing of the blocks. Within the worker thread the blocks will run through a pipeline using the following steps:

1. The block is copied from contiguous memory into a buffer. This buffer is pre-allocated in the

worker thread and retrieved from form a queue of free blocks. At this point the memory in the contiguous buffer can be made available for further DMA transfers by advancing the read pointer.

2. The next step is to decode the block into variable-length packets also known as chunks. Sub-chunks of in previous blocks are prepended to the current sub-chunk to create a data chunk, if possible, or the block is kept to wait for more data to come in.
3. The next action is to gather statistics. This step counts sub-chunks and some other entities processing it into some central data structure, which uses atomic variables to ensure thread synchronization. Histograms are also built using special queues to make sure they have no impact on overall system performance. All this information including configuration of FELIX Core itself is made available to FELIX Monitoring.
4. In this step information is extracted from the chunks. This information is used to route the data packets in the next step. At this time only the E-Link is stored. In future this could be extended to extract an event identifier or other information.
5. The meta information from the previous step is used to assign one or more NetIO clients for the chunk.
6. In this final step the chunk is sent to the NetIO clients assigned in the previous step.

Copies of data, which happen to be expensive, happen in the first and last step of the process. The first step is needed to free up, as fast as possible, the contiguous buffer of memory the DMA writes into. In the last step a copy is needed to transfer data to the underlying network stack. The second copy can be avoided by using NetIO's RDMA capabilities.

For testing purposes the FELIX Card reader can be replaced at runtime by a file reader, which reads data from a file or a data generator, which generates random data. In both cases FELIX Core will see this data as if it was read from a FELIX Card.

6.2.2.2 From Host

Traffic that is sent from NetIO Clients to the frontend is handled by a single thread. Messages from NetIO Clients need to be encoded as 32 bytes including a 2 byte header. The header is decoded and the data is routed to the respective E-Link. HDLC encoding of data is to be done on the NetIO client side with a special library, which is a deliverable of the FELIX project and is available. FELIX Core is not aware of the fact that an E-Link is encoded with HDLC. FELIX Core monitors the number of blocks that are sent to each E-Link. This information is available via the FELIX monitoring tools and the FELIX Core web frontend.

6.2.3 Monitoring

Statistics and configuration information is made available in JSON format by FELIX Core. Internally it runs a small web-server which supplies this information on request. For testing purposes FELIX Core provides a few web pages displaying this information, but as the web server is fairly simple, one should restrict the number of web clients to one or two.

For extensive monitoring of FELIX hosts we have multiple tools:

1. The tool 'felix-mon' to log FELIX data metrics to disk

2. A standalone web application can display current and historical monitoring metrics of FELIX hosts in the same network
3. An integration of FELIX with the TDAQ PBEAST database, that essentially connects FELIX to the TDAQ monitoring infrastructure.

6.3 Performance

TBD - Full-chain measurements ongoing.

6.4 Issues and Improvements

The following issues and possible improvements are known:

1. Busy handling needs to be implemented.
2. Error handling needs to be implemented.
3. Filtering needs to be specified and implemented.
4. E-Link StreamIDs need to be specified and implemented.
5. FELIX Core does not handle configuration, 'elinkconfig' is currently used to do so. We may want to take code from 'elinkconfig' and create a library for configuration. This library can then be used by 'elinkconfig' and FELIX Core for doing the configuration.
6. Lockout may occur if in the FromHost direction large data is sent (split in 32 byte packets) to a single E-Link. As there is currently just one thread handling this direction, we may be spending too much time on large data. We would better use some round-robin algorithm to handle queues from different NetIO subscribers, not to lockout anyone.

:!table: 6

7. NetIO

7.1 Specifications

7.1.1 Communication Patterns

The first communication pattern in NetIO is simple point-to-point communication. In this pattern a sender sends a series of messages to a receiver. The established connection is reliable and preserves order of messages. NetIO sockets are used to establish connections and send messages. A send socket can connect to exactly one receive socket. A receive socket can receive messages from multiple send sockets. The send/receive pattern is illustrated in the following figure.

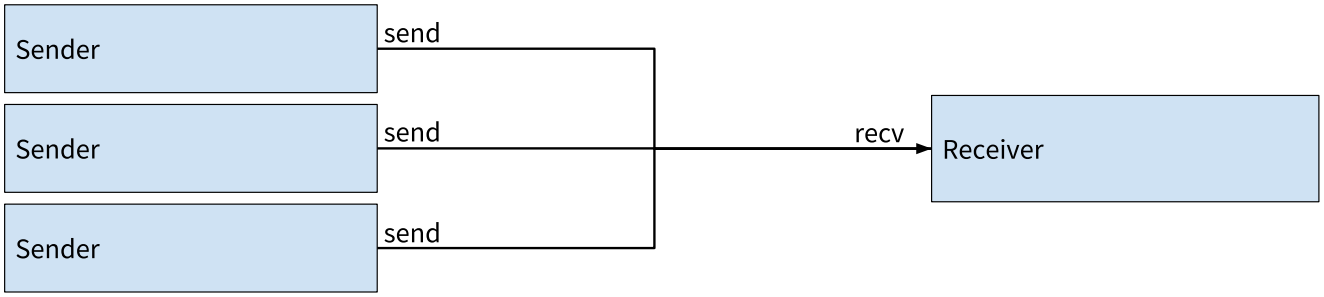


Figure 7. Send/Receive Pattern

The second communication pattern is publish/subscribe communication. A publish/subscribe system, as illustrated in the Figure below, has two types of actors: subscribers, which receive data, and publishers, which publish data. The publisher does not have any prior knowledge of the subscribers. Instead, subscribers send a subscription request to the publishers. Publishers receive the subscription request and add the subscriber to a subscription table. When publishing a message, a publisher looks up the subscription in the subscription table and sends the message to all subscribers. A subscriber can specify in the subscription request a mask to filter the messages that it will receive.

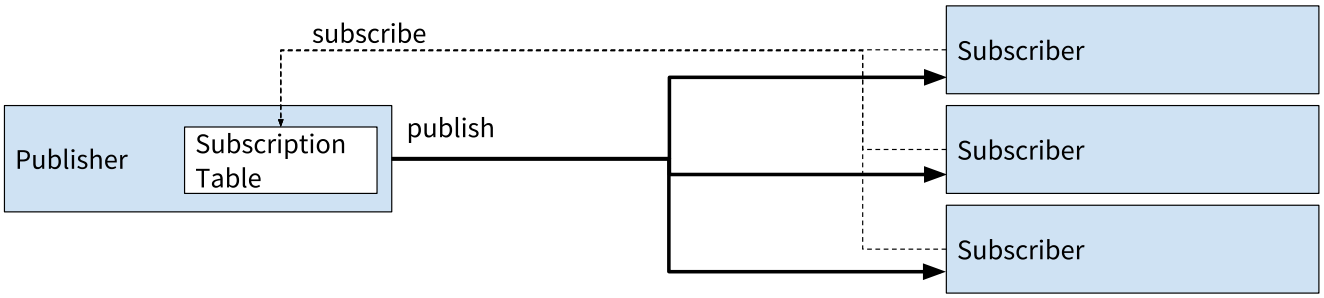


Figure 8. Publish/Subscribe Pattern

7.1.2 Communication Modes

NetIO distinguishes between two principal communication modes: buffered high-throughput communication and unbuffered low-latency communication. The relevant send and receive sockets operate differently in several aspects including the use of threads, use of the event loop, use of callbacks and use of buffers.

7.1.3 Back-End System

NetIO has a back-end system to support different network technologies and APIs. Currently two different back-ends exist. The first back-end uses POSIX sockets to establish reliable connections between endpoints. Primarily this back-end is used on TCP/IP connections in Ethernet networks, but in principle a technology like SDP could also be used. The second back-end is based on libfabric. It is used for communication via Infiniband as well as RDMA-over-Ethernet technologies.

7.2 Implementation

7.2.1 Architecture

The NetIO architecture is illustrated in the following Figure.

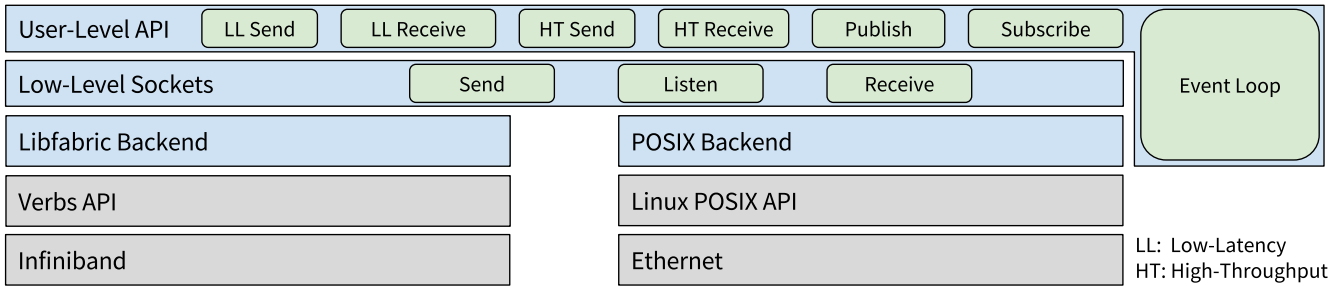


Figure 9. NetIO Architecture

There are two software layers within NetIO. The upper level contains user-level sockets. These are the sockets that application code interacts with. The different socket types are listed in Table~\ref{table:sockets}.

The lower architecture level provides a common interface to the underlying network APIs. The common interface consists of three low-level socket types (a send socket, a listen socket and a receive socket), which are implemented by each back-end. The low-level sockets provide basic connection handling and simple transmission of messages between two endpoints. All higher level functionality like buffering, notification of user code via callbacks, or the publish/subscribe system are implemented in the user-level sockets. This maximizes code sharing among the back-ends, as only code that is specific to the underlying network technology is implemented in the low-level sockets.

Both architecture levels use a central event loop to handle I/O events like connection requests, transmission completions, error conditions, or timeouts. The event loop is executed in a separate thread. Its implementation is based on the epoll framework in the Linux kernel.

Network endpoints are addressed by IP address and port, even for back-ends that do not natively support this form of addressing. For the Infiniband back-end the librdma compatibility layer is used to enable addressing by IP and port.

7.2.2 High-Throughput and Low-Latency Sockets

High-throughout sockets and low-latency sockets work fundamentally different. The differences are described in this section.

A high-throughput send socket does not send out messages immediately but maintains buffers in which messages are copied. The buffers are referred to as pages. Due to the buffering fewer, larger packets are sent on the network link. This approach is more efficient and yields a higher throughput. However, the average transmission latency of any specific message is increased due to the buffering. Once a page is filled the whole page is sent to the receiving end. Additionally, a timer (driven by the central event loop) flushes the page at regular intervals to avoid starvation and infinite latencies on connections with a low message rate. A typical timeout interval is 2\,s. A message is split if it does not fit into a single page. The original message is reconstructed on the receiving side.

A high-throughput receive socket receives pages that contain one or more messages or partial messages. The messages are encoded by simply prepending an 8\,byte length field to the message contents. The high-throughput receive socket maintains two queues: a page queue, which contains unprocessed pages that have been received from a remote, and a message queue, in which messages are stored that are extracted from pages when they are processed. The high-throughput receive socket enqueues received pages in the receive page queue. When user code calls \texttt{recv()} on a high-throughput receive socket, it will return the next message from the message queue. If the message queue is empty, the next page from the receive page queue is processed and the contained messages are stored in the message queue. When processing a received page the contained messages are copied. A new zero-copy mode is currently under development in which it is also possible to avoid this additional copy.

A low-latency send socket does not buffer messages. Messages are immediately sent to the remote process. Unlike for high-throughput send sockets, there is also no additional copy: the message buffer is directly passed to the underlying low-level socket. These design decisions minimize the added latency of a message send operation.

A low-latency receive socket handles incoming messages by passing them to the application code via a user-provided callback routine, instead of enqueuing the messages in a message queue. This approach enables incoming messages to be processed immediately. The message buffer is only valid during execution of the callback. After execution of the callback routine the receive buffer will be freed. If necessary, a user can decide to copy the buffer in the callback routine.

7.3 Performance

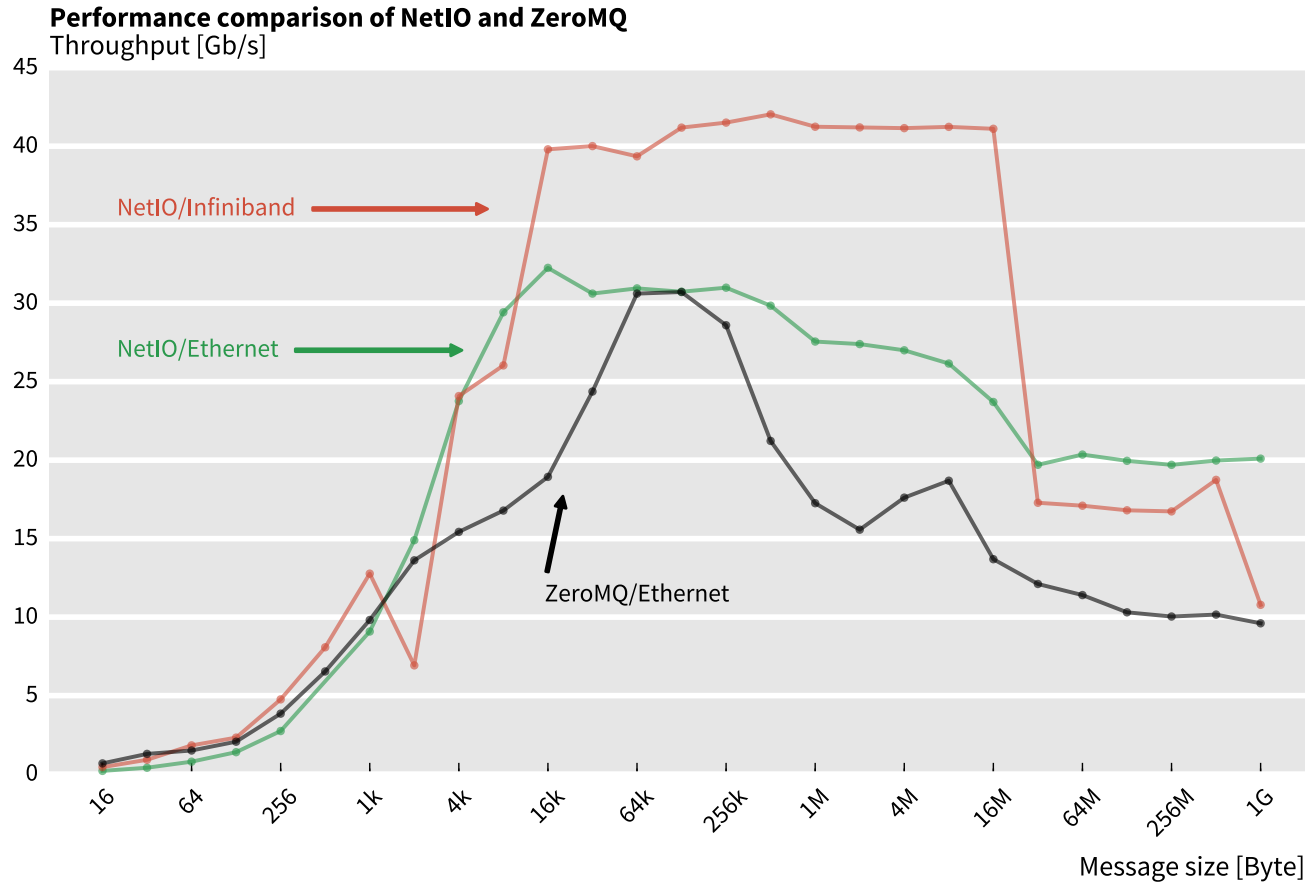


Figure 10. NetIO Throughput

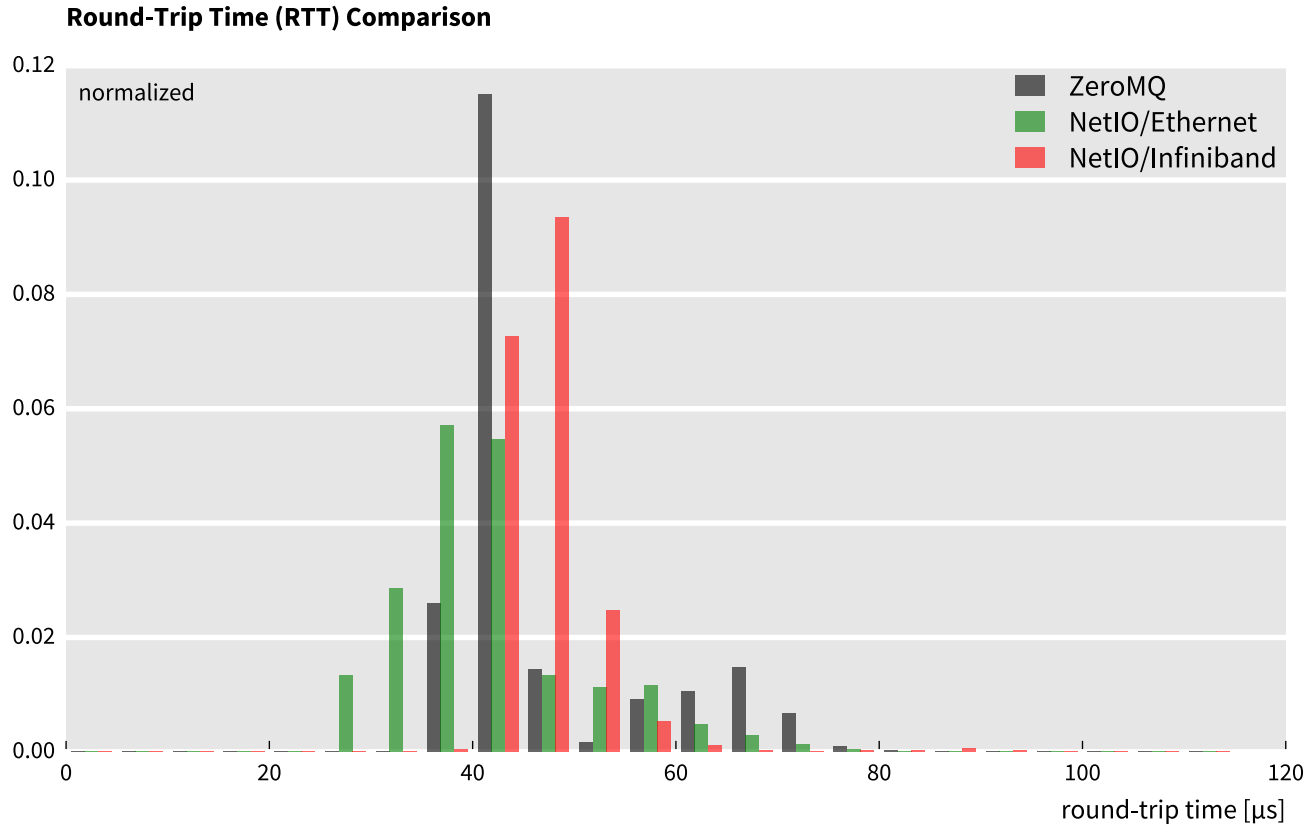


Figure 11. NetIO Round Trip Time

7.4 Issues and Improvements

:!table: 7

Appendix A. FELIX To Host Format

The ToHost format, previously documented under PCIe Bus Data Transfer Format, is a 1024 Byte Block format described below:

A.1 Block Format

A block has a fixed size of 1024 Bytes, including a 4 Byte header. The payload of a block is thus 1020 Bytes long. The block header is defined as follows:

Bits	Length	Description
0-2	3	E-Path
3-5	3	E-Group
6-10	5	GBT
11-15	5	Sequence Number
16-31	16	Start-of-Block Symbol (0xABCD)

```
struct block {
 uint32 elink : 11
 uint32 seqnr : 5;
 uint32 sob : 16;
 char data[1020];
};
```

- The fields GBT, E-Group and E-Path together identify an E-Link.
- The Sequence Number is incremented by one for every block for a particular E-Link. It wraps around after 32 blocks.

A.1.1 Sub-chunk Format

A chunk is an arbitrary length packet, which is packed into blocks. A chunk can span multiple blocks, in which case it is split into sub-chunks. Each sub-chunk has a 2 Byte trailer, but no header. Sub-chunks are Byte-aligned. The trailer format is:

Bits	Length	Description
0-9	10	Length (in bytes)
10	1	CRC Error
11	1	Chunk Error
12	1	Truncation (received more data for this chunk than the maximum chunk length)
13-15	3	Sub-chunk Type

```
struct subchunk_trailer
{
 uint16 length : 10;
 uint16 crc : 1;
 uint16 err : 1
 uint16 trunc : 1
 uint16 type : 3;
};
```

A.1.2 Length

The length of the sub-chunk is specified in Bytes. The length of the trailer is not included. Thus, the maximum length is 1018 Bytes (1020 - 2 Bytes), while the minimum length is 0 Bytes. Padding bytes (see «Padding,padding».) which might occur between sub-chunk data and trailer do not contribute to the sub-chunk length, specified in the length field. If the sub-chunk length is odd, a padding byte is inserted to make it even.

A.1.3 Chunk Error

The error bit is set in case an error has been detected in the current chunk. An error could for example be a CRC error or indicate bad 8B/10B encoding.

CRC errors only are signaled for FULL mode links, as the FULL mode protocol defines a CRC check. CRC errors are indeed signaled with the 'err' bit.

8B/10B encoding errors (two SOPs without EOP after the first SOP or two EOPs without SOP after the first EOP, i.e. framing errors) are signaled as subchunk-type 6.

A.1.4 Truncation

The truncation bit is set when more data for a chunk is received than it has been specified as maximum chunk length.

A.1.5 Sub-chunk Type

The sub-chunk type is a 3-bit encoding according the following mapping:

Name	Value	Description
null	0=0b000	SubChunk is used to fill up a non-full block
first	1=0b001	SubChunk marks the beginning of a chunk
last	2=0b010	SubChunk marks the end of a chunk
both	3=0b011	SubChunk is a whole chunk, i.e. chunk fits into block
middle	4=0b100	SubChunk is a continuation of a previously started chunk, and more SubChunk will follow
timeout	5=0b101	No data received for some given time

Name	Value	Description
error	6=0b110	8B/10B errors -
out-of-band	7=0b111	Out-of-band message. Implies no payload data. The rest of the trailer bits is used to indicate the out-of-band message type.

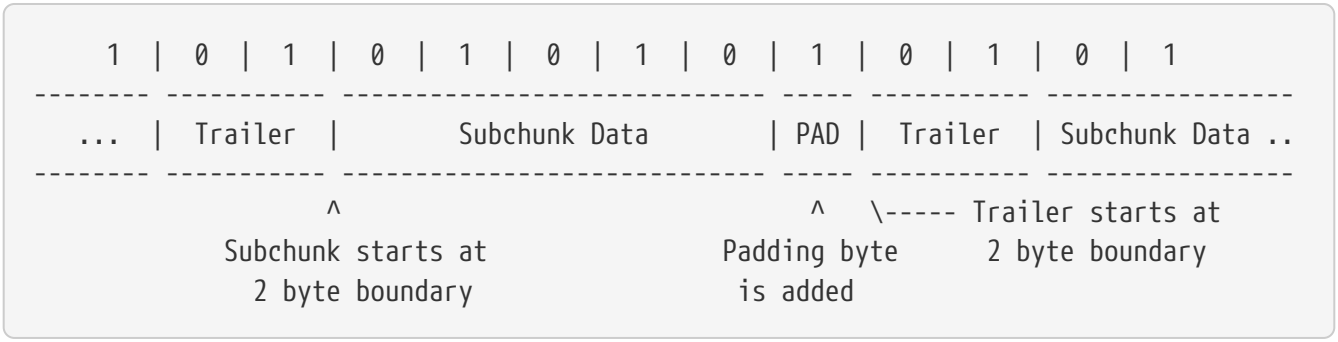
- Out-Of-Band Messages: This type of sub-chunk trailer is used to carry status or error information to the host. No payload data is carried; thus, the implied sub-chunk length is 0. The rest of the bits in the trailer (length field, truncation bit, ...) can be used freely to indicate error types, status codes, etc.

A.1.6 Padding

Since a sub-chunk is at least 2 byte long (sub-chunk trailer size), but the sub-chunk length is not guaranteed to be a multiple of 2 byte, it is not always possible that the block payload can be completely filled with sub-chunks. In that case, padding bytes have to be added.

A.1.6.1 Enforce 16-bit alignment of sub-chunks

Every sub-chunk is placed at a 16-bit boundary. Additionally, every sub-chunk trailer is placed at a 16-bit boundary. In case the sub-chunk length is not a multiple of 2 byte, one padding byte is added between the sub-chunk payload and the sub-chunk trailer:



:!table: 8

Appendix B. FELIX From Host Format

B.1 Formats

Data is sent to the FELIX Card as packets of 32 bytes, starting with a header of 2 bytes, and a payload of 30 bytes.

FromHostHeader, 2 Bytes

Bits	Length	Description
0	1	End-Of-Message, set to 1 for last part of message
1-4	4	Length in 2 byte words
5-7	3	E-Path
8-10	3	E-Group
11-15	5	GBT

- Length is specified in 2 byte words, so length in bytes is always even.

0 :!table: 9

Appendix C. FELIX Input Format

C.1 Formats

Data sent to FELIX over NetIO should start with the following header:

ToFELIXHeader, 16 Bytes

Bits	Length	Description
0-31	32	Length
32-63	32	Reserved
64-95	32	E-Link ID
96-127	64	Reserved

0 :!table: 10

Appendix D. FELIX Output Format

D.1 Formats

Each data chunk sent over NetIO has the following header, followed by the data chunk:

FromFELIXHeader, 8 Bytes

Bits	Length	Description
0-15	16	Length
16-31	16	Status
32-37	6	E-Link ID
38-63	26	GBT ID

References

- [1] FELIX Register Map, https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/regmap/registers.pdf
- [2] JSON Format, http://www.json.org
- [3] FELIX User Manual, https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/FelixUserManual.pdf
- [4] FELIX Driver, https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/felix_driver.pdf
- [5] SWROD, *ATLAS Software ROD*, []()
- [6] ZeroMQ, *Distributed Messaging*, http://zeromq.org
- [7] Zyre, Local Area Clustering for Peer-to-Peer Applications, https://github.com/zeromq/zyre
- [8] Jörn Schumacher, *Utilizing HPC Network Technologies in High Energy Physics Experiments*, available on CDS
- [9] Jörn Schumacher, *Interfacing Detectors and Collecting Data for Large-Scale Experiments in High Energy Physics Using COTS Technology*, PhD Thesis, available on CDS