libexanic is a low-level access library for the ExaNIC hardware. It can be used to transmit and receive raw Ethernet packets with minimum possible latency. Customers who require higher level TCP/IP functionality should instead consider using ExaNIC Sockets.

It is recommended that users first consult the ExaNIC Configuration Guide and ensure that that the ExaNIC software is installed and working.

Opening the device

The first step is to open a handle to the device:

```
#include <exanic/exanic.h>

exanic_t * exanic_acquire_handle(const char *device);
```

The ExaNIC cards in the system will be `"exanic0"`, `"exanic1"`, ... in order of PCI ID. This is the same device name reported by the `exanic-config` utility.

To avoid hardcoding the ExaNIC device name in an application, the following function can also be used to look up the device name and port number from a Linux interface name:

```
#include <exanic/config.h>

int exanic_find_port_by_interface_name(const char *name,
char *device, size_t device_len, int *port_number);
```

`device_len` specifies the length of the `device` buffer into which the device name is returned; it should be at least 8 bytes. The return value is 0 on success and -1 if the requested interface name was not found.

Receive

The ExaNIC delivers packets into logical receive buffers. To attach to a receive buffer, the programmer should call:

```
#include <exanic/fifo_rx.h>

exanic_rx_t * exanic_acquire_rx_buffer(exanic_t *exanic,
             int port_number, int buffer_number);
```

There is no limit to the number of applications that can connect to a receive buffer. By default, all received traffic on a given port will arrive in the buffer obtained by setting `buffer_number = 0`. Frames destined for other hosts are filtered out by hardware unless

promiscuous mode is enabled. When hardware flow steering is enabled, and for `buffer_number > 0`, this function can be used to attach to a userspace flow steering or flow hashing buffer. Flow steering mode configures the ExaNIC hardware to redirect received frames from the default buffer to one of 32 userspace buffers. For details on enabling and configuring flow steering, see the flow steering section of this document.

To receive a frame, call:

```
ssize_t exanic_receive_frame(exanic_rx_t *rx,
char *rx_buf, size_t rx_buf_size,
uint32_t *timestamp);
```

The caller should poll this function in a loop, possibly interspersed with other work. (There is currently no API that would put the current process to sleep waiting for a packet; this may be available in the future, but would naturally involve higher latency.)

The return value will be one of:

`>0`	The length of the frame acquired
`0`	No frame currently available
`-EXANIC_RX_FRAME_ABORTED`	Frame aborted by sender
`-EXANIC_RX_FRAME_CORRUPT`	Frame failed hardware CRC check
`-EXANIC_RX_FRAME_HWOVFL`	Frame lost due to hardware overflow (e.g. insufficient PCIe/memory bandwidth)
`-EXANIC_RX_FRAME_SWOVFL`	Frame lost due to software overflow (e.g. scheduling issue)
`-EXANIC_RX_FRAME_TRUNCATED`	Supplied buffer was too short

The frame delivered to `rx_buf` is a complete Ethernet frame including the CRC footer, however the CRC has already been verified by the hardware.

The frame timestamp is returned via the provided `timestamp` pointer. `timestamp` can be `NULL` if the timestamp is not required. To convert timestamps to real time (nanoseconds since epoch), use:

```
#include <exanic/time.h>

uint64_t exanic_timestamp_to_counter(exanic_t *exanic,
                                     uint32_t timestamp);
```

This function must be called within a few seconds of timestamp acquisition due to the limited precision of the `uint32_t` timestamp; the high order parts of the time are taken from the system clock.

`exanic_receive_frame` may block for a very short period of time (<2 µs) to wait for additional fragments of the frame to arrive via PCI Express. To avoid blocking, there is another function for receiving partial frames:

```
ssize_t exanic_receive_chunk(exanic_rx_t *rx, char *rx_buf,
                             int *more_chunks);
```

This function receives frames in chunks of at most 120 bytes.

When a new chunk is available, `exanic_receive_chunk` will deliver the data to `rx_buf` and return the number of bytes received. `*more_chunks` will be set to `1` if there are more chunks in the frame, or `0` if this is the last chunk. If there is no new data available, `exanic_receive_chunk` will return `0` and `*more_chunks` will be left unchanged.

To receive a complete frame, the caller should poll `exanic_receive_chunk` in a loop until `*more_chunks` is set to `0`.

Transmit

The ExaNIC also contains transmit buffers from which packets are transmitted. To attach to a transmit buffer, the programmer should call:

```
exanic_tx_t * exanic_acquire_tx_buffer(exanic_t *exanic,
          int port_number, size_t requested_size);
```

Each application is allocated a portion of the available transmit buffer space. `requested_size` should be 0 or a multiple of 4096. A value of 0 indicates to use the default size, which is currently 4096. Larger values can increase transmit bandwidth for applications that send many packets close together, at the expense of reducing the number of applications that can share a port.

To transmit a packet, call:

```
int exanic_transmit_frame(exanic_tx_t *tx, const char *frame,
                          size_t frame_size);
```

The frame to be sent should include the Ethernet header but not include the CRC footer which is added by the hardware.

It is also possible to obtain a pointer directly into the NIC frame buffer with:

```
int exanic_begin_transmit_frame(exanic_tx_t *tx, size_t frame_size);
```

Packet transmission is then triggered with:

```
int exanic_end_transmit_frame(exanic_tx_t *tx, size_t frame_size);
```

`frame_size` should be less than or equal to the size allocated in `exanic_begin_transmit_frame`. Care should be taken when using this direct write interface because the returned pointer is in PCI Express memory space; to ensure processor write coalescing it should be written in an approximately sequential fashion and it should not be read from. If in doubt, use the simpler `exanic_transmit_frame` API (published

performance numbers for the ExaNIC use that function, the performance difference is mimimal).

Note: both `exanic_transmit_frame` and `exanic_begin_transmit_frame` may block for a very short period of time (<2 µs) if the transmit buffer is full. Please provide feedback to Exablaze if you require a non-blocking version for your application.

Releasing resources

The API handles should be freed with the following functions:

```
void exanic_release_handle(exanic_t *exanic);
void exanic_release_rx_buffer(exanic_rx_t *rx);
void exanic_release_tx_buffer(exanic_rx_t *rx);
```

Flow steering

Flow steering is available in newer releases of the ExaNIC firmware, which can be obtained from the Exablaze support website. It is supported in firmware versions 2014-05-28 and later. All flow steering is performed in hardware on the ExaNIC, freeing the host CPU to perform other work. Use of ExaNIC flow steering does *not* incur any additional latency penalty on received frames.

Flow steering makes use of the concepts of filters and buffers. Each physical port on the card has a default receive buffer (buffer 0) in host memory to which all traffic is normally delivered and which is shared between the kernel and userspace applications. All ports have an additional 32 userspace buffers, numbered 1 to 32, that can be obtained by a user's application as follows.

First, to attach to an unused receive filter buffer for a given card port, use the function:

```
#include <exanic/fifo_rx.h>

exanic_rx_t * exanic_acquire_unused_filter_buffer(exanic_t *exanic,
                int port_number);
```

This returns an `exanic_rx_t` instance with an unused receive buffer. The `buffer_number` field of this instance indicates which of the 32 userspace buffers was allocated. To obtain a reference to a specific buffer at a later time, use:

```
exanic_rx_t * exanic_acquire_rx_buffer(exanic_t *exanic,
              int port_number, int buffer_number);
```

The `buffer_number` argument should be set to the number of the desired buffer. If this buffer has not yet been allocated it will be allocated by this function. There is no limit to the number of applications that can connect to a given buffer.

Once the application has acquired userspace buffers, it can begin to define rules that direct traffic towards them. Rules exist in two sets, MAC address rules and IP address rules.

Current versions of the firmware supports 128 IP address rules and 64 MAC address rules per physical port, but this may be increased in future firmware revisions.

To define an IP address rule, use the `exanic_ip_filter_t` struct, shown below, setting any fields that you wish to perform a wildcard match on to zero. All fields are stored in network byte order – so you must use appropriate calls to `htons` and `htonl` prior to assignment.

```
#include <exanic/filter.h>

typedef struct exanic_ip_filter
{
  uint32_t    src_addr;   /**< Source IP address of packet */
  uint32_t    dst_addr;   /**< Destination IP address of packet */
  uint16_t    src_port;   /**< Source port of packet */
  uint16_t    dst_port;   /**< Destination port of packet */
  uint8_t     protocol;   /**< IPPROTO_UDP or IPPROTO_TCP */
} exanic_ip_filter_t;
```

To bind the rule to the previously obtained buffer use the function `exanic_filter_add_ip`, passing in the previously obtained buffer and the filter rule you have defined:

```
int exanic_filter_add_ip(exanic_t *exanic,
                         const exanic_rx_t *buffer,
                         const exanic_ip_filter_t *filter);
```

This function will return a unique positive integer ID for each rule created on a given port, or -1 if no further rules can be allocated. From this point on, frames matching the given rule will be steered by the hardware to the defined userspace buffer, and will not reach the Linux network stack or the default userspace buffer. Calls to `exanic_receive_frame`, using the acquired buffer as the first argument, will only return frames matching the rules associated with the buffer. Frames not matching any rules will still be delivered to buffer 0.

To add a MAC address rule, first define the rule using the `exanic_mac_filter_t` struct, where all fields are in network byte order. This means that `dst_mac[0]` is the most significant byte of the destination MAC address. Set to zero any field to perform a wildcard match.

```
typedef struct exanic_mac_filter
{
    uint8_t     dst_mac[6];
    uint16_t    ethertype;
    uint16_t    vlan;
    int         vlan_match_method;
} exanic_mac_filter_t;
```

In the case of VLAN tagged frames, a number of match modes can be used. These are defined in the enumeration `vlan_match_method`:

```
#include <exanic/pcie_if.h>

 enum vlan_match_method
 {
     /** Match on all frames, whether VLAN or not. */
     EXANIC_VLAN_MATCH_METHOD_ALL        = 0,

     /** Only match on the VLAN given. */
     EXANIC_VLAN_MATCH_METHOD_SPECIFIC   = 1,

     /** Only match if frame does not have a vlan tag. */
     EXANIC_VLAN_MATCH_METHOD_NOT_VLAN   = 2,

     /** Match frames that have a VLAN tag, but not those that don't. */
     EXANIC_VLAN_MATCH_METHOD_ALL_VLAN   = 3,
 };
```

Then, bind the filter to a previously acquired buffer using:

```
int exanic_filter_add_mac(exanic_t *exanic,
                          const exanic_rx_t *buffer,
                          const exanic_mac_filter_t *filter);
```

This function will return a unique positive integer ID for each MAC address rule on a given port, or -1 if no hardware rules remain. As in the case of IP rules, once a rule is bound to a buffer, calls to `exanic_receive_frame`for that buffer will only return frames matching rules associated with that buffer.

It should be mentioned that it is possible to create rules such that more than one rule matches an incoming ethernet frame. When this is the case, priority is resolved as follows:

- MAC address rules have higher priority than IP address rules,
- Within a given set of MAC or IP address rules, the lower the rule ID, the higher the rule priority.

Rules can be released using:

```
int exanic_filter_remove_ip(exanic_t *exanic,
                            int port_number,
                            int filter_id);

int exanic_filter_remove_mac(exanic_t *exanic,
                             int port_number,
                             int filter_id);
```

Flow hashing

The ExaNIC also supports hardware flow hashing which can be used to distribute load evenly amongst a number of host processors. When enabled, the ExaNIC hardware will deliver received frames to one of a number of userspace receive buffers based on a hash computed over IP headers. The hash function guarantees that all frames belonging to a given flow will always arrive in the same buffer. For a given ExaNIC port, flow steering and flow hashing cannot be used at the same time.

To enable flow hashing on a specific port, use the function:

```
#include <exanic/fifo_rx.h>
int exanic_enable_flow_hashing(exanic_t *exanic, int port_number,
                           int max_buffers, int hash_function);
```

This function will allocate up to `max_buffers` userspace receive buffers and distribute IP traffic evenly among them using the `hash_function`. The number of buffers must be a power of 2, and currently a maximum of 32 buffers are available. The number of actual receive buffers allocated is returned by this function. The available hash functions are defined in `rx_hash_function`.

```
enum rx_hash_function
{
  /* Symmetric hash over source port, destination port */
  EXANIC_RX_HASH_FUNCTION_PORT = 0,

  /* Symmetric hash over source ip, destination ip */
  EXANIC_RX_HASH_FUNCTION_IP = 1,

/* Symmetric hash over source ip & port, destination ip & port */
  EXANIC_RX_HASH_FUNCTION_PORT_IP = 2,
};
```

The application can then attach to the userspace flow hashing buffers by calling `exanic_acquire_rx_buffer`, passing in buffer numbers from 1 to the number of allocated userspace receive buffers. IP traffic will be balanced across each of these buffers, and `exanic_receive_frame` can be used to obtain the traffic in each buffer.

To disable flow hashing and free all allocated buffers, use:

```
void exanic_disable_flow_hashing(exanic_t *exanic,
                                 int port_number);
```