CS–1997–21

Trapeze API

Jeffrey S. Chase
Andrew J. Gallatin
Alvin R. Lebeck
Kenneth G. Yocum

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

November 1997

Trapeze API

Jeff Chase, Andrew Gallatin, Alvin Lebeck, and Ken Yocum

Department of Computer Science
Duke University

Revision 5.0

This document defines the programming interface (API) of the Trapeze messaging system, which supports high-speed point-to-point communication in Myrinet networks. The Trapeze API emphasizes performance over ease-of-use; although it can be used directly by applications, it is intended as a substrate for higher-level messaging packages, e.g., DSM or RPC systems.

1 Introduction

Trapeze is a messaging facility for high-speed point-to-point communication in computer clusters. The current Trapeze implementation consists of three components:

1. ***Trapeze-MCP***, a firmware program (a Myrinet Control Program or MCP) for Myrinet LANai 4.x network interface cards (NICs) using the Peripheral Component Interconnect (PCI) I/O bus standard. Since the Trapeze-MCP firmware program defines the behavior of the NIC hardware and its interface to the host, we often refer to the adapter and its firmware together as the *Trapeze network interface* or just "the NIC". The important features of the Trapeze network interface are described in papers available through the Trapeze web site.
2. ***Trapeze-API***, a collection of macros and library procedures to send and receive messages using the Trapeze network interface. The Trapeze-API runs on the host, linked into application processes and/or the operating system kernel. The purpose of this document is to define the Trapeze-API programming interface that allows user application code or kernel code to communicate across the network using Trapeze.
3. ***Trapeze Network Driver***, a kernel-based Myrinet device driver for AlphaStations running Digital Unix 4.x and Intel platforms running FreeBSD 2.2.x. The driver allows qualified application programs to access the NIC directly without the overhead of kernel intervention (after an initial setup). It also allows TCP/IP communication through the Berkeley Unix socket system calls, by making the Trapeze NIC appear as an ordinary (Ethernet-like) network device to the kernel-based TCP/IP protocol stack — thus the driver is often referred to as the "TPZ/IP" driver.

The Trapeze distribution includes directions for building and installing the Trapeze-API libraries and Trapeze-MCP firmware, building and booting a kernel with a TPZ/IP driver, and setting up routing tables and device files for the network. Once Trapeze has been properly installed and configured, kernel or application code may access the network using the programming interface defined in this document.

This document is the authoritative reference on the Trapeze programming interface. Trapeze continues to evolve and some of the details of the interface and its implementation will change, but the definition is reasonably stable. However, the Technical Report version of this document may be out-of-date; the most recent version of the Trapeze release and this document is always available through the Trapeze web site (*http://www.cs.duke.edu/ari/trapeze*), which also includes research papers on Trapeze with comparisons to related work.

Paragraphs formatted like this present additional details that are implementation-dependent and/or likely to change in a future implementation. These details are provided to clarify what is really going on, or to document limitations and "features" of the current prototype.

2 Basic Concepts and Terminology

Trapeze is a *memory-mapped network interface*: the entire NIC memory is visible at fixed locations in a region of the host physical address space, and a host CPU uses the network interface by executing instructions that read and write memory locations on the NIC using programmed I/O. Trapeze can be used either by the kernel or by user programs:

- An authorized user program can use Trapeze by mapping a portion of the NIC memory into its process virtual address space using an *mmap* system call on a device file for the network interface. It can then send and receive messages without the overhead of system calls.
- The kernel can use Trapeze by addressing the NIC memory directly, using kernel virtual addresses bound to the NIC physical addresses in a machine-dependent way. The kernel may also request interrupts to notify it of incoming messages.

Currently Trapeze can be used by only a single user program at a time, and it cannot be used from kernel space if it is also being used from user space. If a user program initializes the interface while the kernel is using it, the results are undefined and may include a system crash or data corruption. If Trapeze is used by the kernel, the superuser should set permissions on the /dev/myri device file to exclude user programs from accessing the interface.*

This document uses the term *client* to refer to the software module using Trapeze. The client can be an application program, a user-level system library (e.g., a distributed shared memory system), or a kernel module. In the kernel, there may be several protocol modules using Trapeze at the same time, e.g., a kernel-kernel RPC package and the network driver for TCP/IP traffic. To identify the protocol module, each message is tagged with a *header ID* packet type field (*tpz_hid_t*) at the sender, and demultiplexed at the receiver as described below. The header ID is ignored for user-user communication.

Figure 2-1and Figure 2-2 depict the use of Trapeze from kernel space and user space respectively. The various elements of these figures are discussed in the following subsections, which describe the key concepts of the Trapeze interface:

Trapeze endpoint data structure. Any Trapeze client (kernel protocol module or user process) accesses the Trapeze interface by linking with the Trapeze-API library and calling the procedures and macros defined in this document. The Trapeze-API code linked into the client interacts with the firmware through

a shared *endpoint* data structure in NIC memory. See Section 2.1.

Control messages. The Trapeze interface is designed to allow short messages (*control messages*) to be sent and received by directly accessing small message buffers in the endpoint structure on the NIC. The Trapeze-API primitives for constructing and decoding messages are designed to allow the implementation to send and receive control message contents using either programmed I/O or DMA, depending on performance characteristics of the host processor and bus architecture. See Section 2.2.

Payloads. A control message may include an optional attached *payload* with additional data. Payloads are always sent and received using DMA for the highest bandwidth and lowest host CPU overhead. Items larger than the maximum control message size are always sent and received as payloads. A Trapeze receiver must designate buffer space for incoming payloads. See Section 2.3.

Payload Tokens. Trapeze *payload tokens* allow a message sender to specify a buffer in the receiver's memory where the receiving NIC is to deposit a payload, with the consent of the receiver. Payload tokens are useful for early demultiplexing of incoming payloads, to eliminate copying overhead. They were designed to allow zero-copy handling of replies in request/response message exchanges (e.g., RPC), but they can also be used for other forms of memory-to-memory block data transfer. See Section 2.4.

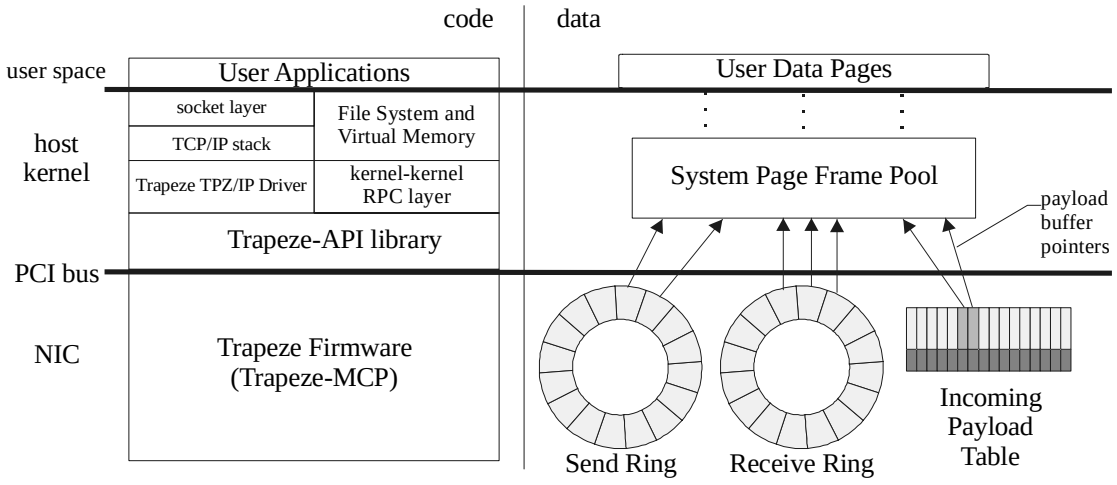

Figure 2-1: A Trapeze endpoint used by the kernel for distributed services and TCP/IP networking.

2.1 Trapeze Endpoint and Message Rings

A Trapeze endpoint consists of three data structures on the NIC shared memory:

- The *send ring*, a circular producer/consumer queue of outgoing messages. Each ring entry corresponds to a single outgoing message, and includes a small fixed-size message buffer, and related per-message state (e.g., transmission status).

- The *receive ring*, a second producer/consumer ring of incoming messages. Each ring entry corresponds to a single message, and includes a fixed-size message buffer and related per-message state.
- The *incoming payload table (IPT)*, a table of pointers to host buffers preallocated for specific incoming messages tagged with payload tokens (Section 2.4).

From the point of view of the host, the NIC produces incoming messages in the receive ring and consumes outgoing messages in the send ring. Each entry in the send ring or receive ring contains control information, including busy and free bits used by the Trapeze-API routines and the Trapeze-MCP firmware to synchronize use of the ring entries between the host and the NIC.

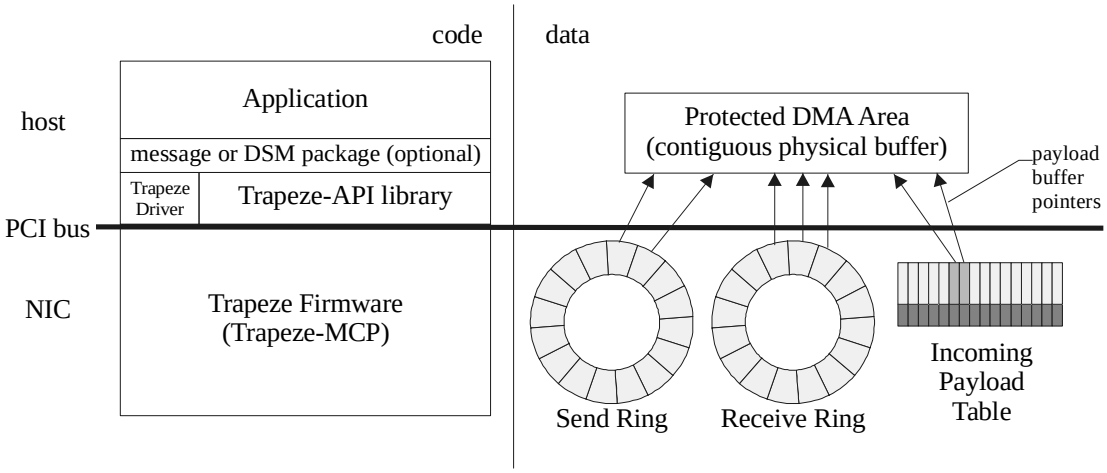

Figure 2-2: A Trapeze endpoint used directly by a user-space application.

To send or receive a message, the client calls an API *get* primitive to obtain a pointer to an entry in one of the rings, operates on it to form or decode the message, and calls a *release* primitive to indicate that processing is complete. In between the *get* and the *release*, the client is said to *hold* the ring entry. Failure to release a held ring entry in a timely fashion will lock up the NIC. Figure 2-3 depicts the operation of the receive ring; the send ring is similar.

On the sending side, the *get* primitive (*tpz_get_sendmsg*) returns the next free entry in the send ring, i.e., the next ring entry that is not held and does not have a pending message yet to be transmitted. With bursty send traffic, the send ring may fill up with pending messages due to Myrinet's link-level flow control, thus the caller may need to spin or wait for a free send ring entry. Once a ring entry is acquired, the holder then constructs a message as described below, specifies a destination, and releases the entry with *tpz_release_sendmsg*, which instructs the NIC to send the message if it is valid. The caller may invalidate the message with *tpz_abort_sendmsg* before releasing it, in case it detects some error before releasing the message.

On the receiving side, the *get* primitive (*tpz_get_rcvmsg*) returns the receive ring entry containing the next undelivered incoming message. If no undelivered messages exist, the caller may spin or wait for an incoming message. Once the host obtains an incoming message, it calls the release primitive (*tpz_release_rcvmsg*) when it is finished processing the message, freeing the ring entry to receive another message.

2.2 Ring Entries and Control Messages

In addition to control and status fields, each entry in the send and receive ring includes a small inline *message buffer*, a fixed-size array of words for message contents. The data sent and received through the inline buffer in the ring entry is called a *control message*. A control message may be up to *TPZ_CTRLMSG_WORDS* 32-bit words in length.

.

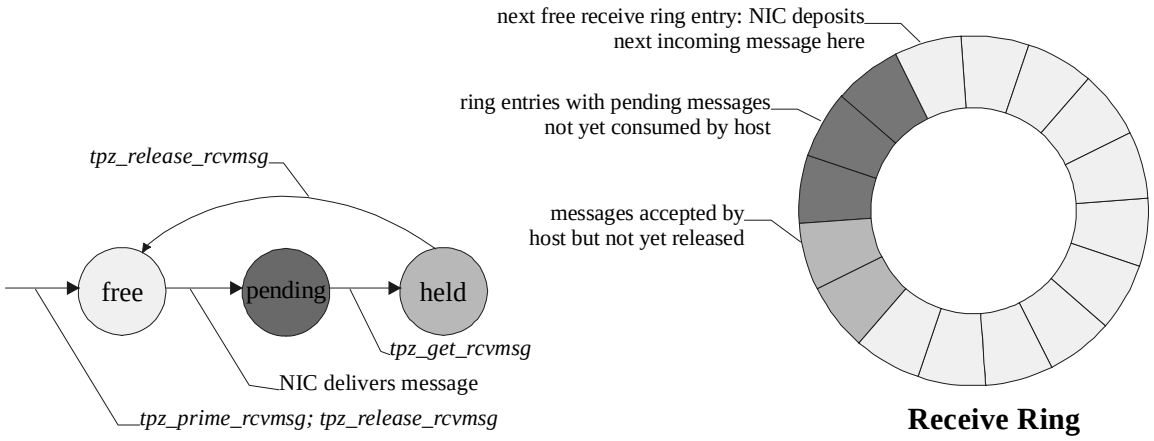

Figure 2-3: States and transitions of entries in the Trapeze receive ring.

The *get* primitives return a pointer to a ring entry as a *tpz_msgbuf_t*, which the client passes back as an argument to other API methods to operate on the ring entry and its message buffer. Ordinarily, the client uses *tpz_reserve* to "push" data items onto an outgoing control message, and *tpz_scan* to "pop" data items from an incoming control message, as described in Section 9 and illustrated in Table 9-5.

> ***Implementation note****: Currently, a tpz_msgbuf_t points directly to the ring entry in NIC memory, including four words of Trapeze-defined packet header (header, type, destination, control message length, and payload length). There is currently space for 30 words (120 bytes) of user data. Many operations in the Trapeze API convert a tpz_msgbuf_t pointer into a ring index with a shift/mask, then use the index to access internal data structures kept in host memory rather than NIC memory. Loads and stores on NIC memory are minimized to reduce host CPU overheads for programmed I/O.*

2.3 Payloads and Payload Buffers

The holder of a ring entry (*tpz_msgbuf_t)* may attach an optional *payload buffer* to the ring entry. The payload buffer is a region of host memory that holds extra data (the *payload*) sent or received with an outgoing or incoming control message. Payloads are always carried in the same network packet as the associated control message, and are always moved between host memory and the NIC using DMA. The maximum payload size is *TPZ_PAYLOAD_WORDS* 32-bit words; Trapeze can be used to send variable-length messages by specifying the length of the attached payload, for a maximum total message size of *TPZ_CTRLMSG_WORDS* + *TPZ_PAYLOAD_WORDS* 32-bit words. Section 9 defines the primitives for operating on payload buffers.

> *The maximum payload size (TPZ_PAYLOAD_WORDS) is a build-time parameter in the Trapeze-MCP firmware. It is merely a convention between the sender and receiver; the Myrinet hardware does not define a maximum packet size. However, the maximum payload size is constrained by the amount of buffer memory on the NIC.*

A payload buffer pointer is an instance of type *tpz_payload_t*. On all current Trapeze platforms, the *tpz_payload_t* must be a word-aligned address of a physically contiguous region of host memory. For in-kernel use, we typically use aligned page frames for all payload buffers, and configure the maximum payload size to be equal to the virtual memory page frame size; kernel modifications not included in the Trapeze release manage physical memory as a unified pool of page frames shared between the network subsystem and the file and virtual memory subsystems. In user space, payload buffers are allocated from a large, pinned, physically contiguous host buffer assigned to the process by the Trapeze network driver when the device (endpoint) is attached to the process using the **mmap** system call. This fixed buffer region is variously called the *copy block, protected DMA area*, or *communication segment* in other user-level messaging systems that use a similar technique.

> *Currently, the interface for converting virtual buffer addresses into DMA addresses is non-uniform between user-mode Trapeze and kernel-mode Trapeze. For kernel use on the Alpha systems, the tpz_payload_t is a KSEG kernel virtual address, and the conversion to a valid DMA address occurs within the Trapeze-API routines. In user space, it is the client's responsibility to convert virtual buffer pointers to DMA addresses by adding an offset to the DMA area base returned by the driver at bind time. We plan to clean this up by providing a new API macro to convert between virtual addresses and tpz_payload_t DMA addresses.*

Attaching a payload to a send ring entry causes the NIC to send the contents of the payload buffer with the message when the message is released. On the receive side, the NIC receives the message into the next free receive ring entry, and deposits the payload in the payload buffer attached to that ring entry (unless the message is tagged with a payload token as described in Section 2.4). The receiving host may detach the payload from the ring entry (using *tpz_detach_rcvmsg*) for further processing. For example, an in-kernel Trapeze client might detach a payload buffer and map it into a user address space, in order to deliver received data to a user process without copying it.

It is important for the client to keep the receive ring properly stocked with payload buffers at all times. If the receive ring entry selected by the NIC for an incoming message and payload does not have a payload buffer attached, then the NIC will silently drop the payload. All entries in the receive ring should be initialized with payload buffers using *tpz_prime_rcvmsg* and *tpz_attach_rcvmsg* before enabling the interface

(see Section 5). If the host detaches a received payload from its ring entry then it must attach a fresh payload buffer before releasing the ring entry to receive another message.

2.4 Payload Tokens

Payload tokens provide a means for a sender (say, site **A**) to direct the NIC to deposit a payload at a specific location in the memory of a receiver (say, site **B**), with the receiver's consent. First, **B** must designate the memory region as a payload buffer using *tpz_get_token*, which returns the payload token as a *tpz_token_t*. **B** then communicates the payload token to **A** by some means, e.g., by sending it in a message to **B**. Once **A** possesses a valid payload token for the desired memory region in **B**, it can transfer a payload directly to the designated buffer by sending a message to **B** with an attached payload, and tagging the message with the payload token when the message is sent with *tpz_release_sendmsg*. Payload tokens may be cancelled at any time by the receiver (**B**) using *tpz_cancel_token*. Section 8 defines the primitives for using payload tokens.

Payload tokens use the *incoming payload table (IPT)* in the endpoint structure in NIC memory (see Section 2.1). The IPT is an array of slots containing pointers to the payload buffers associated with the currently outstanding payload tokens. The payload token consists of an IPT slot index and a *key* value selected by Trapeze as minimal protection against forged or dangling payload tokens. The NIC changes the key value in the IPT slot as soon as it receives a message carrying a valid payload token for the slot; thus each payload token may be used at most once.

> *Currently there only a few hundred IPT entries available, but this is not a fundamental limitation. We are also considering an extension that allows a payload token to be used multiple times, with a requirement that the sender and receiver synchronize (e.g., through a barrier) before the next message tagged with that payload token is sent. We have also discussed an extension that would allow the sender to specify an offset from the base of the payload buffer named by the payload token; this would be very close to Hamlyn's sender-based memory management.*

3 Overview of Trapeze-API Types and Calls

Table 3-1 lists the client-visible data types defined in Trapeze-API header files and used in the Trapeze programming interface, and Table 4-2 lists the API procedures and macros, broken down by category and use (sender or receiver).

Table 3-1: Defined types for Trapeze API.

Type Identifier	Type	Description
tpz_hid_t	*int*	Header ID used to demultiplex messages on the receiver. Each Trapeze message includes a *tpz_hid_t* field in a system-defined header. This field identifies the protocol module (e.g., an RPC messaging package or TCP/IP driver) for the message. Unique *tpz_hid_t* values are assigned to protocol modules at initialization time, according to conventions defined outside of Trapeze.
tpz_msgbuf_t	*address*	Pointer to a control message object. Pointers of type *tpz_msgbuf_t* are generated and used only by API methods (see Section 2.2).
tpz_token_t	*int*	Payload token identifier (see Section 2.4).
tpz_msgspec_t	*int*	Message specifier for an outgoing message, passed as an argument to *tpz_release_sendmsg*, and identifying the type of message being sent (control message, control message with payload, or control message and payload tagged with a payload token).
tpz_payload_t	*address*	Payload buffer pointer (see Section 2.3). A *tpz_payload_t* must be a word-aligned host DMA address of a physically contiguous buffer in host memory.

4 Interface Initialization

int tpz_ready(tpz_hid_t hid)

Return true if the interface is loaded and initialized, else return false. If the *hid* argument is nonzero, tpz_ready will not return true unless a receiver handler has been registered for the specified *hid* using *tpz_register_rcv_handler.*

int tpz_init(tpz_hid_t hid);

Initialize the messaging subsystem. Returns true for success, false for failure. In user space, *tpz_init* loads the Trapeze-MCP from the file whose pathname is referenced by a global variable *mcp_filename*, in accordance with a Myricom convention (see Table 5-3). For in-kernel use, the Trapeze network driver loads the MCP at configuration time; if the LANai load was not successful then *tpz_init* returns false. The hid argument specifies the client module that is ready to send and receive messages through Trapeze.

Currently the hid argument is ignored from user space, and should be zero. In user space, tpz_init discards all previous interface state, if any. See tpz_disable below.

void tpz_enable()

Start the NIC. When *tpz_enable* returns, Trapeze may deliver incoming messages. All receiver initialization must be completed before calling *tpz_enable* (see Section 5).

void tpz_disable()

Disable the NIC. This discards all pending messages, calls the I/O completion routines for all pending transmissions, and puts the interface in a state that consumes and discards all incoming or outgoing messages. Once disabled, the interface may be initialized and re-enabled in the usual fashion. *Not implemented.*l

Table 4-2: Trapeze API Methods.

	Send	**Receive**
initialization/control	*tpz_init, tpz_ready, tpz_enable, tpz_status*	
receive initialization	*tpz_register_rcv_handler, tpz_remove_rcv_handler, tpz_rcv_ring_size, tpz_prime_rcvmsg, tpz_attach_rcvmsg*	
getting ring entries	*tpz_get_sendmsg, tpz_get_sendmsg_spinwait*	*tpz_get_rcvmsg, tpz_get_rcvmsg_spinwait*
control messages	*tpz_mtod, tpz_attach_iocomplete, tpz_rereserve, tpz_reserve*	*tpz_mtod, tpz_scan, tpz_rescan*
payloads	*tpz_attach_sendmsg, tpz_set_payload_len*	*tpz_detach_rcvmsg, tpz_get_payload_len*
payload tokens	*tpz_get_replytoken, tpz_cancel_replytoken, tpz_release_replytoken*	
releasing	*tpz_senderr, tpz_aim, tpz_release_sendmsg*	*tpz_release_rcvmsg*
miscellaneous	*tpz_check_xmit_complete, tpz_send_ring_size*	*tpz_rcv_ring_size*
user-supplied	*io_completion_t*	*rcvintr_t*

int tpz_status()

Return NIC status. Any nonzero value is a Trapeze-MCP failure code, indicating that the firmware has crashed and the NIC must be disabled and reinitialized before the network can be used again.

extern int tpz_enabled

Trapeze-API maintains an externally visible up/down flag called *tpz_enabled*. A value of zero means the interface is down: receive interrupts should be ignored, and no sends or receives should be attempted.

5 Receiver Initialization

*typedef tpz_msgbuf_t *(rcvintr_t)(tpz_msgbuf_t, tpz_hid_t);*
void tpz_register_rcv_handler(rcvintr_t)

Register a receive handler for the higher-level protocol module with the specified *tpz_hid_t* ID. Registering a handler requests Trapeze to upcall the handler from the device receiver interrupt routine on each receive interrupt. Section 7 defines the interface and operation of receive handlers.

*Currently, receive handlers may be registered only from within the kernel, and every client module in-kernel **must** register a receive handler. Message reception in-kernel is always by interrupts. Message reception in user space is always by polling, thus tpz_register_rcv_handler has no effect in user space.*

int tpz_rcv_ring_size();

Return the number of entries in the receive ring.

tpz_msgbuf_t tpz_prime_rcvmsg();

Get a pointer to the next receive ring entry. This is used only during initialization while the interface is disabled. The intent is that the caller will use *tpz_prime_rcv_msg* to obtain pointers to receive message buffers in order to attach payload buffers with *tpz_attach_rcvmsg* before enabling the interface. Every ring entry returned by *tpz_prime_rcvmsg* must be released with *tpz_release_rcvmsg*.

```
int ring_size, i;
tpz_msgbuf_t msg;

/* Note: in kernel space the mcp_filename is not needed, and hid is meaningful. */
tpz_hid_t hid = 0;                              /* header ID is ignored in user space */
char* mcp_filename = "/usr/project/trapeze/mcp/mcp.dat";

tpz_init(hid);                                  /* attach and initialize interface */

/* prime the receive ring with payload buffers generated by my_get_payload_buffer() */
ring_size = tpz_rcv_ring_size();
for (i = 0; i < ring_size; i++) {
        msg = tpz_prime_rcvmsg();
        tpz_attach_rcvmsg(msg, my_get_payload_buffer());
        tpz_release_rcvmsg(msg);
}

tpz_enable();                                   /* start the interface */
```

Table 5-3: Initializing a Trapeze interface and priming the receive ring.

6 Sending Messages

int tpz_send_ring_size();

Return the number of *tpz_msgbuf_t* entries in the send ring.

int tpz_send_allowed();

Return true if there is a free send entry, false if no entry is free. Under load, the send ring may be filled with pending sends, e.g., if the send engine is blocked due to Myrinet's link-level flow control.

tpz_msgbuf_t tpz_get_sendmsg();
tpz_msgbuf_t tpz_get_sendmsg_spinwait();

Acquire and return a send ring entry. For *tpz_get_sendmsg*, return null if there are no free entries in the send ring. For *tpz_get_sendmsg_spinwait*, spin wait for a predetermined backoff if no send ring entry is available. The spin variant is suitable for calling in a tight loop, but *tpz_get_sendmsg* is not.

Currently the backoff is very long, so everything locks up if the interface is blocked.

void tpz_senderr(tpz_msgbuf_t);

Mark a held send ring entry as an aborted send. This procedure is used when the sender decides to abort a send after allocating a send ring entry. The client must release the entry with *tpz_release_sendmsg*, but the NIC will not attempt to send the message.

tpz_release_sendmsg(tpz_msgbuf_t, int dest, tpz_msgspec_t specifier);

Release a held send ring entry. This marks the message as "ready" so the NIC send code may consume it and advance beyond the entry in the send ring. The NIC processes the entry by sending the message it contains, if and only if the message was not marked as an aborted send by *tpz_senderr.* If the message was aborted with *tpz_senderr*, the NIC skips it and advances to the next send ring entry.

The target node (*dest*) is specified as a small integer chosen from a compact set of unique node numbers, suitable for indexing into a NIC routing table (see addressing/routing notes on the Trapeze web site). The *specifier* is either a message type or payload token (see Section 2.4). If the message has no attached payload, then it is a pure *control message* and the specifier is TPZ_CTRL. If the message has an payload attached but no payload token, the specifier is TPZ_UNSOL (indicating an "unsolicited" payload).

int tpz_sendindex(tpz_msgbuf_t)

Return the send ring index of the specified *tpz_msgbuf_t.*

7 Receiving Messages

tpz_msgbuf_t tpz_get_rcvmsg();
tpz_msgbuf_t tpz_get_rcvmsg_spinwait();

Get a *tpz_msgbuf_t* handle for the next incoming message. For *tpz_get_rcvmsg* return null if there are no undelivered messages to receive. For *tpz_get_rcvmsg_spinwait*, spin wait until a message arrives.

We should have a spinwait variant that returns quickly, but is suitable for calling in a tight loop.

tpz_msgbuf_t tpz_peek_rcvmsg();
void tpz_consume_rcvmsg();

Acquire a receive message in two steps equivalent to a *tpz_get_rcvmsg*. For *tpz_peek_rcvmsg*, return a *tpz_msgbuf_t* handle to the next undelivered ring entry, but do not transition the entry to the held state, i.e., the next *tpz_get_rcvmsg* will return the same message. A peeked-at message can be acquired with *tpz_consume_rcvmsg*.

These methods are used in combination for interrupt-driven receives. The Trapeze receive handler peeks at the next received message, then passes it to the registered handler matching the header ID. The registered handler uses *tpz_consume_rcvmsg* to consume the message, then peeks at the next message. If the next message's header ID does not match the current handler, it returns the header ID back to the Trapeze receive handler, which passes the unconsumed ring entry to the correct handler. In this way all undelivered received messages are consumed on each receive interrupt.

Need to explain more clearly why we don't just use tpz_get_rcvmsg here.

tpz_release_rcvmsg(tpz_msgbuf_t);

Release a held receive ring entry. This marks the received ring entry as available to receive a new message. The user must release each received ring entry after message processing is complete. Failure to release any receive ring entry will deadlock the NIC.

*typedef tpz_hid_t *(rcvintr_t)(tpz_msgbuf_t, tpz_hid_t);*
tpz_msgbuf_t client_receive_handler(tpz_msgbuf_t, tpz_hid_t)

A Trapeze client may define a receiver handler routine and register it with *tpz_register_rcv_handler.* The Trapeze network driver receive interrupt handler upcalls the client's handler if (but not only if) a message has been received with a header ID associated with the registered handler. The handler is expected to "peek" and acquire (consume) one or more messages in the receive ring, consuming all consecutive received messages with a matching header ID. If it encounters a message with a different header ID, it should return the *tpz_hid_t* for the unconsumed ring entry. If no unconsumed messages remain, it should return TPZ_NO_HID.

The current scheme requires the client to look directly at the message ring fields from the tpz_msgbuf_t. It should use a macro to get a pointer to the header, so we can hide the details of how data comes off the ring.

```
tpz_hid_t
tpz_null_rcv_handler(tpz_msgbuf_t msg, tpz_hid_t unit) {
        do {
                /* msg is for us: consume it and release it */
                tpz_consume_rcvmsg();
                printf("consuming message for hid %d\n", MY_HID);
                tpz_release_rcvmsg(msg);

                /* peek at the next one and loop if there's something for us */
                if (msg = tpz_peek_rcvmsg())
                        hid = ntohl(msg->pkt_type);/* byte swap may be needed */
        } while (msg && (hid == MY_HID));
        return (msg ? hid : TPZ_NO_HID);
}
```

Table 7-4: A null Trapeze receive handler for header ID MY_HID.

void tpz_slow_enqueue(tpz_msgbuf_t msg)
tpz_msgbuf_t tpz_slow_dequeue()

Trapeze-API provides an auxiliary queue in host memory for receive messages if needed by the client. In the kernel, it is convenient to queue messages that are consumed off of the NIC receive ring by an interrupt handler, but for which processing must be deferred to an ordinary process or thread. If the queue is shared with an interrupt handler in this fashion then calls to these routines must block out interrupts, e.g., using *splimp()*.

8 Handling Payload Tokens

tpz_token_t = tpz_get_token(tpz_payload_t buf);

Allocate a slot in the incoming payload table (IPT) for the specified payload buffer, and return a payload token for the buffer. If the IPT is full, return *TPZ_REPLY_NOSLOT.*

See Section 2.4 for a description of payload tokens and their use. It is the client's responsibility to synchronize calls to *tpz_get_token*, and to release the IPT slot with *tpz_release_token* after receiving a message tagged with the payload token. Also, the payload buffer must be big enough to receive any payload sent to it; if it is too small, the NIC will write payload data beyond the end of the buffer, possibly corrupting an unrelated data item in host memory.

Trapeze protects against damage from dangling payload tokens in the following way. Payload tokens returned by *tpz_get_token* include both an IPT slot number and a *key.* The NIC matches the payload token key of an incoming reply against a key value in the IPT. If the key does not match, the NIC simply drops the incoming payload, and delivers the control message without the associated payload. In this case, it is the client's responsibility to recognize the control message as incomplete and/or outdated e.g., by examining a request sequence number in the message, and/or using *tpz_get_payload_len* to determine if the payload was discarded.

For safety, the NIC cancels the key when it receives a message tagged with the payload token. This protects the payload buffer from being overwritten by a second message (e.g., a duplicate) after the first message has been delivered. Also, a payload token may be cancelled at any time with *tpz_cancel_token* (e.g., if the client abandons waiting for an expected message due to a timeout). In either case, the IPT entry remains disabled until it is reused for a future *tpz_get_token* request.

tpz_payload_t tpz_release_token(tpz_token_t token)

Release the IPT slot for a payload token, returning the *tpz_payload_t* of the payload buffer named by the payload token. The returned value is just for sanity testing. The client must call *tpz_release_token* after receiving a message tagged with the payload token; the client must recognize for itself when such a message has been received.

tpz_payload_t tpz_cancel_token(tpz_token_t token)

Cancel a payload token. The client uses *tpz_cancel_token* when it intends to reuse the payload buffer named by token for some other purpose, eventhough the expected message has not been received. It might be used if, for example, some error condition detected after the IPT entry was allocated has aborted the request message, or if a request message was sent, but the sender is no longer expecting a reply, e.g., due to a timeout. *tpz_cancel_token* is identical to *tpz_release_token*, except that *tpz_cancel_token* writes the NIC IPT entry to cancel it, rather than assuming that it has been cancelled by the NIC on receiving a message tagged with the payload token.

9 Message Functions

caddr_t tpz_reserve(tpz_msgbuf_t, int len);
caddr_t tpz_scan(tpz_msgbuf_t, int len);

Return a pointer to the message data for a ring entry. Use *tpz_reserve* to “push” data into an outgoing message buffer, and *tpz_scan* to “pop” data from a receive buffer. Both functions maintain (in host memory) a sliding offset field for each message buffer; the offset starts at the beginning of the message and increments by *len* for each call to *tpz_reserve* or *tpz_scan*. These functions check for overflow of the message buffer. Table 9-5 illustrates the use of these and other message functions.

The nature of the pointer returned by *tpz_scan* or *tpz_reserve* is implementation-dependent. It may be a direct pointer to a NIC control message buffer, or it may point to a shadow buffer in host memory. If a shadow buffer is used, the message data is transferred to and from the NIC using either DMA or PIO, depending on which is most efficient for the specific architecture and message size.

caddr_t tpz_rereserve(tpz_msgbuf_t);
caddr_t tpz_rescan(tpz_msgbuf_t);

Return a pointer to the start of the message data for a ring entry. The pointer returned is identical to the pointer returned by the first call to *tpz_scan* or *tpz_reserve* on that message buffer. This is useful, for example, if items have been scanned off of a message, but some piece of client code needs a pointer back to the beginning of the message. These variants are used to reduce the need to pass message data pointers around the client call chain.

caddr_t tpz_mtod(tpz_msgbuf_t);

Message to data: return the address of the start of the NIC control message buffer in a ring entry. This is a more direct but less flexible alternative to *tpz_reserve* and *tpz_scan*, on either the sending or receiving side. The pointer returned by *tpz_mtod* can be used to read or write message data directly using programmed I/O, bypassing any shadow buffer maintained by the Trapeze-API implementation. With *tpz_mtod*, the client has complete control over the host memory references used to access the buffer. Use of *tpz_mtod* is not recommended when *tpz_scan* or *tpz_reserve* are used on the same message.

void tpz_set_ctlmsg_len(tpz_msgbuf_t, int len);
int tpz_get_ctlmsg_len(tpz_msgbuf_t)

Get/set the length of the control message portion of a message. This is done automatically when *tpz_scan* and *tpz_reserve* are used, but the client must do it when *tpz_mtod* is used. **Note**: the length is measured in 32-bit words.

void tpz_attach_rcvmsg(tpz_msgbuf_t, tpz_payload_t);

Attach a payload buffer to a receive ring entry. The NIC uses the payload buffer to hold the payload (if any) attached to the next message received into that receive ring entry, assuming the message is not tagged with a payload token designating an alternative payload buffer. The NIC assumes that payload buffers attached to receive ring entries are large enough to receive the largest allowable incoming payload (*TPZ_PAYLOAD_WORDS* 32-bit words). If a payload buffer is too small for an incoming payload, the NIC may deposit incoming message data beyond the end of the payload buffer, possibly leading to unexpected host behavior.

typedef void (io_completion_t)(int, int, caddr_t);
void tpz_attach_sendmsg(tpz_msgbuf_t, tpz_payload_t, io_completion_t, caddr_t arg);
void tpz_check_xmit_complete();

Attach a payload buffer to a send ring entry. The buffer contains a payload to send with the outgoing message when it is released; *tpz_set_payload_len* MUST be called to specify the length of the attached payload before the message is released.

Trapeze-API will call the specified *io_completion_t* handler routine with the specified *arg* at some time after the transmit completes, in order to notify the user that it is safe to overwrite the payload buffer. Trapeze saves the handler pointer and argument in host memory until the transmit is complete. The Trapeze network interface provides no transmit-complete interrupts, so the notifications of completed sends may be delayed. Trapeze-API calls the handler when either: (1) *tpz_get_sendmsg** reuses a send ring entry with a pending *io_completion_t*, or (2) the client calls *tpz_check_xmit_complete*. For *tpz_check_xmit_complete*, call the completion routines for every completed send ring entry with a pending *io_completion_t*. Note: since the completion routine may be called from *tpz_get_sendmsg*, it should not block; send completion routines in the kernel should raise the IPL appropriately if any in-kernel Trapeze module might request a send from an interrupt handler.

For historical reasons I/O completion functions take three arguments. The first two are always zero. The third argument is the arg passed to tpz_attach_sendmsg when the transmitted payload was attached.

tpz_payload_t tpz_detach_rcvmsg(tpz_msgbuf_t)
tpz_payload_t tpz_detach_sendmsg(tpz_msgbuf_t)
tpz_payload_t tpz_attached(tpz_msgbuf_t)

Return the payload buffer attached to this ring entry, if any, else return null. For the *detach* variants, detach the payload buffer before returning it.

If a payload buffer is detached from a receive ring entry, a fresh buffer should be attached before releasing the entry, to supply space for any payload on the next message received into the ring entry. **Note**: the presence of a payload buffer attached to the ring entry for a received message does not indicate that a payload was received with the message, unless *tpz_get_payload_len* returns a nonzero value. If no payload was received, or if the received payload was not detached, it is not necessary to attach a fresh payload buffer to the receive ring entry before releasing it; Trapeze will continue to use or reuse the payload buffer already attached to the ring entry.

It is not particularly useful to detach a payload from a send ring entry, as the payload attached to an outgoing message is implicitly detached when the message is sent, and the client is notified by an upcall to an *io_completion_t*.

void tpz_set_payload_len(tpz_msgbuf_t, int)
int tpz_get_payload_len(tpz_msgbuf_t)

Use these procedures to get (from an incoming message) or set (for an outgoing message) the length of the payload attached to the message. **Note**: the length is measured in 32-bit words.

```
void sender(tpz_payload_t payload)
{
        tpz_msgbuf_t msg;
        struct abc *p;

        /* get a message buffer...don't bother with failure cases for this example */
        msg = get_sendmsg_spinwait();

        /* place data (1...2...3...4...5...6) in arbitary structs in the control message */
        p = (struct abc*)tpz_reserve(sizeof(struct abc));
        p->a = 0; p->b = 2; p->c = 3;
        p = (struct abc*)tpz_reserve(sizeof(struct abc));
        p->a = 4; p->b = 5; p->c = 6;

        /* attach 512 bytes of payload; must be a valid tpz_payload_t */
        tpz_attach_sendmsg(msg, payload, 0, 0);
        tpz_set_payload_len(msg, 512 >> 2);

        /* send the message to node #2 */
        tpz_release_sendmsg(msg, 2, TPZ_UNSOL);
}

tpz_payload_t receiver()
{
        tpz_msgbuf_t msg;
        tpz_payload_t payload_data;
        struct abc *p;

        /* get next msg...don't bother with failure case for this example */
        msg = get_rcvmsg_spinwait();

        /* read data from the message */
        p = (struct abc*)tpz_scan(sizeof(struct abc));
        printf("%d, %d, %d\n", p->a, p->b, p->c);/* 1, 2, 3 */
        p = (struct abc*)tpz_scan(sizeof(struct abc));
        printf("%d, %d, %d\n", p->a, p->b, p->c);/* 4, 5, 6 */

        /* detach payload so we can return it; restock with fresh buffer */
        assert(tpz_get_payload_len(msg) << 2 == 512);
        payload_data = tpz_detach_rcvmsg(msg);
        tpz_attach_rcvmsg(msg, my_get_payload_buffer());
        tpz_release_rcvmsg(msg);

        return(payload_data);
}
```

Table 9-5: A simple send/receive example illustrating use of the message operations.