Overview of UNH EXS 1.3.0 for Programmers

Robert D. Russell
Patrick MacArthur
InterOperability Laboratory
University of New Hampshire
Durham, New Hampshire 03824
{rdr,pmacarth}@iol.unh.edu

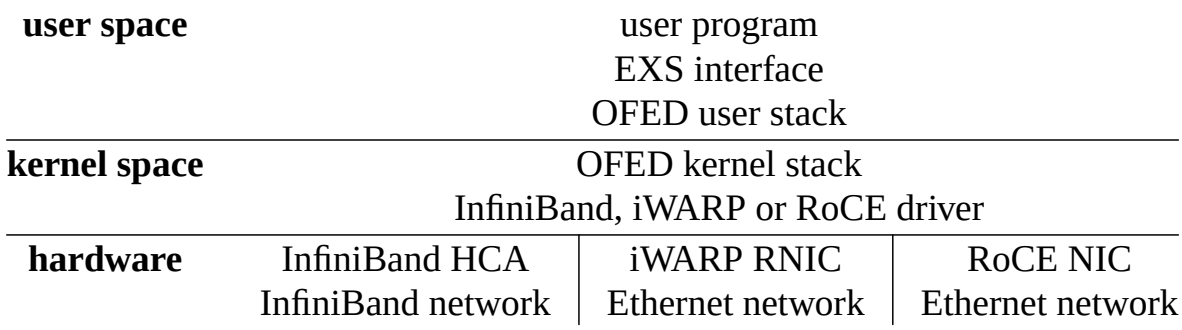
1. Introduction

The **Extended Sockets API** (ES-API) is a specification published by the **OpenGroup** that defines extensions to the traditional socket API in order to provide asynchronous I/O and also memory registration for Remote Direct Memory Access (RDMA). These two major new features enable programmers to take advantage of today's multi-core processors and RDMA network hardware, such as InfiniBand, iWARP and RoCE interfaces, in a convenient yet efficient manner.

This document describes version 1.3.0 of the UNH EXS interface, an extended implementation of the ES-API. Section 11 outlines the differences between version 1.3.0 and the ES-API standard.

The UNH EXS interface provides most of the features specified in the ES-API, and provides a few additional features that give the programmer more flexibility. For example, the programmer can choose to program with synchronous rather than asynchronous I/O, and/or to program with or without memory registration. The programmer can also conveniently "tune" certain aspects of the EXS interface to take advantage of application requirements in order to provide better performance.

In addition, the UNH EXS interface is designed to be implemented entirely in user space on the Linux operating system. This latter feature provides easy porting, modification and adoption of EXS, since it requires no changes to existing Linux kernels. The current implementation of the EXS interface is based on the **OpenFabrics Enterprise Distribution** (OFED) (recently renamed to be the **OpenFabrics Software** (OFS)), a free software package provided by the **OpenFabrics Alliance** (OFA). The OFS runs in both user and kernel space, and provides efficient, asynchronous access to various types of RDMA networking hardware, currently InfiniBand, iWARP and RoCE. The layering structure of this system is shown below. The term **RNIC** stands for "RDMA Network Interface Card", which is the term used by iWARP for the hardware interface card that enables a computer to use zero-copy transfers over Ethernet fibers and cables. **HCA** stands for "Host Channel Adapter", which is the term used by InfiniBand for the hardware interface card that enables a computer to use zero-copy transfers over InfiniBand fabric. RoCE does not seem to have a special name for it's interface card. We will use the term **Channel Adapter** (**CA**) to refer generically to any of the 3 hardware interface cards.



The principal difference between the UNH EXS interface and the OpenGroup's ES-API is that the UNH EXS interface runs entirely in user space. Therefore, it is not integrated with "normal" kernel sockets. This means EXS functions cannot be used with "normal" sockets, and EXS sockets cannot be used with "normal" socket functions. This, in turn, means the UNH EXS interface had to add numerous functions, such as **exs_socket**(), **exs_bind**(), etc., that the ES-API does not define because the ES-API expects to be implemented in the kernel and integrated with "normal" sockets. The ES-API is intended to be an extension to existing "normal" sockets and therefore does not need to redefine "normal" functions.

Another difference is that the UNH EXS interface is designed as a "thin", efficient layer between the user and the Channel Adapter via the OFED stack. It therefore retains much of the packet orientation of the underlying protocols for both **SOCK_STREAM** and **SOCK_SEQPACKET** sockets.

This document gives an overview of the general concepts of EXS, and how the UNH EXS interface can generally be used by a programmer. It is not a reference manual.

Section 2
: starts by giving an overview of the style of programming for which the features offered by EXS are most suitable. Subsequent topics include EXS asynchronous I/O, dealing with event queues using the **exs_qcreate**(), **exs_qdelete**() and **exs_qdequeue**() functions, and the need for memory registration when using RDMA.

Section 3
: describes how a user program initializes the EXS environment with the **exs_init**() function.

Section 4
: describes how users create and manage EXS client and server connections by using the **exs_socket**(), **exs_bind**(), **exs_connect**(), **exs_listen**(), **exs_accept**(), and **exs_close**() functions.

Section 5
: describes how users transfer data over EXS connections by using the **exs_send**() and **exs_recv**() functions.

Section 6
: describes the internal flow control mechanisms utilized by the UNH EXS interface and their effects on performance.

Section 7
: describes some ways EXS performance can be tuned through the use of the **exs_fcntl**() function.

Section 8
: describes ways to utilize registered and unregistered memory by using the **exs_mregister**() and **exs_mderegister**() functions, and the **EXS_MHANDLE_UNREGISTERED** value.

Section 9
: describes ways to utilize synchronous and asynchronous I/O and the various flags that control this.

Section 10
: gives a series of sample client programs showing how to convert from a client using normal sockets (with no access to RDMA hardware) to a client using EXS sockets (with full access to RDMA hardware) in asynchronous mode and explicit memory registration.

Section 11
: describes the current status of UNH EXS 1.3.0 and the differences between it and the ES-API standard.

2. Overview of the EXS style of programming

The general paradigm for programming with EXS is “threads” and “events” programming. Although it is not necessary to use threads in order to use EXS, it is the simplest way to utilize the parallelism provided by multi-core processors and to take advantage of the asynchronous I/O facilities in EXS. Threads are also utilized internally by the UNH EXS interface.

2.1. Asynchronous I/O

EXS accomplishes asynchronous I/O by partitioning the user’s interaction with I/O via the EXS interface into two distinct phases: the “start” phase, and the “completion” phase. Each asynchronous operation starts with a start phase. If that phase fails, then there is no subsequent completion phase — instead, the user must deal with the error. But if the start phase succeeds, the asynchronous EXS operation proceeds in parallel with independent processing by the user thread that started it. The completion phase begins when the user calls a separate function that waits for the asynchronous operation to finish. This function returns detailed results about the asynchronous operation that can indicate either success or failure of the operation.

In EXS, the start phase of an asynchronous operation is accomplished by new EXS functions with names similar to the standard socket and UNIX I/O function calls. These include: **exs_accept**(), **exs_close**(), **exs_connect**(), **exs_recv**(), and **exs_send**(). These functions first verify that their user-supplied parameter values make such an operation possible. They then cause the EXS interface to “start” an operation but do not wait for the operation to actually take place. Instead they immediately return to the user after giving the OFED stack all the information necessary to proceed with the operation in parallel with (i.e., asynchronously to) the user’s threads.

As a parameter to these new asynchronous EXS functions the user must specify a pointer to a “queue” object previously created by the user (as discussed next in this section). When the asynchronous operation completes, either successfully or not, the EXS interface will “post a completion event” to that queue. This “event” is a structure which contains detailed information

about the success or failure of the operation. The user obtains this information by "dequeuing" the event structure from the queue. If the EXS interface posts the event before the user tries to dequeue it, the queue object stores the information until the user performs the dequeue operation. If the user tries to dequeue an event before the asynchronous operation is finished, the dequeue operation will block until either the EXS interface posts the event or an amount of time specified by the user as a parameter to the dequeue operation elapses. This can be an indefinite amount of time. There are 3 possible scenarios, as illustrated in the following diagrams.

Step	**User thread**	**EXS interface**
1	call asynchronous function	
2		check parameters
3		return error
4	deal with error	

Start phase results in an error, I/O not started.

Step	**User thread**	**EXS interface**
1	call asynchronous function	
2		check parameters
3		start I/O operation
4		return ok
5	proceed with other work	proceed with I/O operation
6	proceed with other work	post I/O completion event
7	proceed with other work	
8	dequeue I/O completion event	
9	process I/O completion event	

Start phase ok, user thread calls **exs_qdequeue**() after I/O is complete.

Step	User thread	EXS interface
1	call asynchronous function	
2		check parameters
3		start I/O operation
4		return ok
5	proceed with other work	proceed with I/O operation
6	dequeue I/O completion event	proceed with I/O operation
7	wait for I/O completion event	proceed with I/O operation
8	wait for I/O completion event	post I/O completion event
9	process I/O completion event	

Start phase ok, user thread calls **exs_qdequeue**() before I/O is complete.

2.1.1. Creating an event queue

A user creates an event queue by using:

qhandle = exs_qcreate(depth);

where **depth** specifies the guaranteed minimum number of events that the user wants to be able to store in this queue. This function dynamically allocates memory to hold at least that number of event structures. A user can create numerous different event queues, but these are valid only in the context of the calling process (and its threads) — these event queues are not valid in any child processes forked by this parent. The value returned in **qhandle** will be NULL if there was an error of some sort, in which case the error code is stored in the global **errno**. Otherwise, the value returned in **qhandle** is a "handle" (i.e., a pointer) of type **exs_qhandle_t**, which must be passed as a parameter in subsequent EXS function calls that start asynchronous operations. This handle identifies the queue to which an asynchronous operation will post its completion event.

2.1.2. Deleting an event queue

A user deletes an event queue by using:

result = exs_qdelete(qhandle);

where **qhandle** identifies a previously created event queue. The value returned in **result** will be 0 if the call to **exs_qdelete**() was successful, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno**.

Note that any completion events posted to, but not yet dequeued from, this queue will be lost when this function is called. Also note that any outstanding asynchronous operations that reference this queue and that have been started but not yet completed will cause the call to **exs_qdelete**() to fail.

2.1.3. Using an event queue

Following a successful call to one or more EXS asynchronous functions, the user must use:

nevents = exs_qdequeue(qhandle, event_vector, count, timeout);

to wait for those EXS asynchronous functions to complete. In the call to **exs_qdequeue**(), **qhandle** must be the same value used previously in the EXS asynchronous function call that started an operation, **event_vector** must be a user-provided array big enough to hold **count** events of type **exs_event_t**, and **timeout** is a pointer to a structure of type **struct timeval** containing the maximum amount of time the user wants to wait for an event to happen. The **timeout** parameter can be NULL if the user wants to wait indefinitely. This call to **exs_qdequeue**() will block until either the **timeout** elapses or at least one event is posted to the queue and is copied into **event_vector**. The value returned in **nevents** will be 0 if a timeout occurred, a positive value if **nevents** events were removed from the queue and copied into **event_vector**, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno**.

If the **nevents** value returned by **exs_qdequeue**() is positive, then each of the first **nevents** structures of the **event_vector** array will have been filled in by the EXS interface with information to identify the operation that posted the event and to convey back to the user the final results of that operation. For each of these structures in the **event_vector** array, the following fields will be filled in as follows:

exs_evt_errno
: is 0 if the asynchronous I/O operation completed successfully, otherwise it contains an error code (i.e., a Linux **errno** value) to indicate why it failed.

exs_evt_socket
: is a copy of the **fd** parameter value used in the EXS asynchronous function to identify the connection.

exs_evt_ahandle
: is a copy of the **ahandle** parameter value used in the EXS asynchronous function. This value is chosen by the user for identification purposes only, and is opaque to (i.e., not used by) the EXS interface.

exs_evt_type
: is a constant value indicating which type of EXS asynchronous function caused the event. These values will be discussed below with the individual EXS asynchronous functions. Examples include **EXS_EVT_ACCEPT**, **EXS_EVT_CLOSE**, **EXS_EVT_CONNECT**, **EXS_EVT_RECV** and **EXS_EVT_SEND**.

exs_evt_union
: contains a structure whose type depends on the **exs_evt_type** value. The fields in these structures will be discussed below with the individual EXS asynchronous functions.

2.2. Memory registration

In order to utilize the direct memory-to-memory transfer feature of the RDMA interface hardware, the CA requires that the virtual memory involved on both ends of a transfer be “registered”. This registration accomplishes several necessary functions:

(1) It establishes the location and size of a memory area to be utilized in RDMA transfers;

(2) It establishes the read/write/execute permissions granted to both the local and remote CAs for accessing that memory area;

(3) It “pins” the user’s virtual memory area into real (i.e., physical) memory so that a CA can access the memory without going through the CPU’s paging hardware.

Note that any type of virtual memory can be registered — stack memory, heap memory, global memory, etc. Common usage will most likely be to register dynamically allocated memory, but since this is not necessary, EXS does NOT allocate the memory it is registering.

The mechanism by which a user explicitly registers and deregisters memory is described in section 8 below. That section also describes a means by which the user can request dynamic registration and deregistration of memory as part of an individual EXS function call. A word of warning, however: dynamic registration and deregistration is an expensive procedure, and should be used only as a temporary "bridge" when converting an existing application from "normal" sockets to EXS. The benefits of EXS require explicit memory registration by the user program.

3. Establishing the EXS environment

Unlike with normal sockets, a program that wants to utilize EXS must explicitly initialize the EXS environment by using:

result = exs_init(version);

where **version** is the version of the EXS interface the user wishes to use. The current version is given by the global constant **EXS_VERSION**, which is defined in the file **exs.h** which, in turn, must be included by all programs wishing to utilize EXS. The **result** value is 0 if the exs_init() was successful, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno**.

This function will initialize the EXS interface for the calling process. It must be performed exactly once before any other EXS function is called, and is usually placed near the beginning of program execution.

4. Managing an EXS connection

As with classic sockets, there are three different types of connections a user can create:

(1) a client connection

(2) a server listening post

(3) a server agent connection

4.1. Creating an EXS socket

In order to establish either a client connection or a server listening post, the user must first set up the local endpoint for the connection by using:

fd = exs_socket(domain, socket_type, protocol);

where the value of **domain** must be **PF_INET** or **PF_INET6** (or equivalently, **AF_INET** and **AF_INET6**), **socket_type** must be either **SOCK_STREAM** or **SOCK_SEQPACKET**, and **protocol** must be **IPPROTO_TCP** (or equivalently, **0**). This is a synchronous operation, so there is no event associated with its completion. The value returned in **fd** will be -1 if there was an error of some sort, in which case the error code is stored in the global **errno**; otherwise, the value returned in **fd** is a non-negative integer **file descriptor** which must be used to identify this connection in all subsequent calls to EXS functions.

As of version 1.3.0, a large intermediate receive buffer will be allocated for **SOCK_STREAM** sockets unless this feature is turned off via the interfaces described in section 7. The size of this buffer defaults to 3 Gigibytes, and may also be modified via the interfaces described in section 7.

Which EXS functions to call after a successful return from **exs_socket**() depends on whether the user wants to establish a client connection or a server listening post.

4.2. Establishing an EXS client connection

The client end of a connection is the simplest to establish, although this operation will not succeed unless the remote server's listening post has been previously established (discussed in section 4.3.1). Creation of a client connection is started by:

result = exs_connect(fd, server_address, server_addrlen, connect_flags, timeout, qhandle, ahandle);

This function is normally the next EXS function called by a client after a successful call to **exs_socket**(), and **fd** identifies the endpoint established by the **exs_socket**(). **server_address** is a pointer to a structure of type **struct sockaddr** into which the user has stored the IPv4/IPv6 address and port number of the server's listening post. **server_addrlen** is the size in bytes of the structure pointed to by **server_address**. The value of the **connect_flags** parameter is usually 0, but see below for the other possibilities. **timeout** is a structure of type **struct timeval** which specifies the maximum amount of time the client thread is willing to wait for the **exs_connect**() to complete successfully. If the **timeout** pointer is NULL, **exs_connect**() will wait indefinitely. **qhandle** identifies an event queue that will be used to wait for the completion of this **exs_connect**(), and **ahandle** is an arbitrary pointer value chosen by the user which will be returned in the **exs_evt_ahandle** field of the event notification (discussed previously in section 2.1.3).

The value returned in **result** will be 0 if the **exs_connect**() was successfully started, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno** and no asynchronous activity is started.

If the **exs_connect**() started successfully, it will operate asynchronously with the user thread that started it. During this time the user should not call any additional EXS functions for this connection, because the internal state of the connection will be undefined until the **exs_connect**() has completed and the EXS interface has posted to the user's **qhandle** an event whose **exs_evt_type** field contains the value **EXS_EVT_CONNECT**. Once this event has been received, the user is able to use the connection to transmit data to and from the remote server.

The only **connect_flags** value currently defined for **exs_connect**() is **EXS_BLOCK**. When this flag value is present, the value of the **qhandle** and **ahandle** parameters are ignored by the EXS interface, and can be NULL. The **EXS_BLOCK** flag value should be supplied when the user wants the **exs_connect**() to operate synchronously rather than asynchronously.

4.3. Establishing a server EXS connection

Establishing a server is a bit more complex than establishing a client. As with regular sockets it involves two distinct sequences.

4.3.1. Establishing a server listening post

A **listening post** is the socket endpoint set up by a server in order to allow clients to be able to contact the server. To enable such contact, a listening post must be bound to an IPv4/IPv6 address and port number pair that is known to clients. This IPv4/IPv6 address and port number pair is analogous to a "1-800" number set up by a business — it must be known to customers who will use it to contact the business by telephone.

The first step in establishing a listening post is to create the socket endpoint using the **exs_socket**() function, as already discussed in section 4.1. The next step is to bind that socket endpoint to the IPv4/IPv6 address and port number pair which will be utilized by clients in the **exs_connect**() function, as just discussed in section 4.2. This binding is done by using:

result = exs_bind(fd, server_address, server_addrlen);

The **fd** parameter identifies the endpoint established by a previously successful call to **exs_socket**(). **server_address** is a pointer to a structure of type **struct sockaddr** into which the user has stored the IPv4/IPv6 address and port number to be assigned by the EXS interface to the server, and **server_addrlen** is the size in bytes of the structure pointed to by **server_address**. This is a synchronous operation, so there is no event associated with its completion. The value returned in **result** will be 0 if the call to **exs_bind**() was successful, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno**.

After a successful call to **exs_bind**(), the next step in establishing a listening post is to identify the socket to the EXS interface as a listening post and to establish a **backlog** for it. This **backlog** is analogous to establishing the maximum number of calls which can be kept waiting on a 1-800 line, and is accomplished using:

result = exs_listen(fd, backlog);

fd has the same value as that used in the previous call to **exs_bind**(), and **backlog** is the maximum number of client connections that can be "kept on hold" until a server connection dedicated to a new client can be set up (i.e., until a customer can be switched to a free agent in the 1-800 call center). This is a synchronous operation, so there is no event associated with its completion. The value returned in **result** will be 0 if the **exs_listen**() was successful, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno**.

At this point the server process is ready to somehow convey the IPv4/IPv6 address (or DNS name) and port number of its listening post to potential client processes all over the world. Well established servers, such as the World Wide Web HTTP service, have been assigned "Well Known Ports" by IANA (the Internet Assigned Numbers Authority) (e.g., port 80 has been assigned to the HTTP service), so this knowledge is available to every browser in the world, and programmers only have to know the IPv4/IPv6 address (or DNS name) of their destination in order to connect their client process to an HTTP server. Most programmers do not have the luxury of working with Well Known Ports. Therefore, the means by which the server's IPv4/IPv6 address and port number are made known to clients is outside the scope of EXS.

4.3.2. Accepting connections on the server

Once a server has established a listening post, it needs to set up to accept connections from clients. This is analogous to hiring agents to answer the phones in the 1-800 call center. When a

business answers a call to the 1-800 number, the call is switched to a separate line into a call center where a single agent deals exclusively with that individual customer — the 1-800 number remains ready to accept new calls and to switch them to other agents. Similarly, when the listening post accepts a connection from a client, it creates a new socket that will deal exclusively with that client — the listening post itself remains ready to accept new connections, but never actually transfers any data with any clients. The listening post is set up to do this using:

result = exs_accept(fd, address_vector, count, accept_flags, qhandle);

The **fd** parameter is the same as the one used in a previously successful call to **exs_listen**(), **address_vector** is a pointer to an array of structures of type **exs_acceptaddr**, and **count** is the number of elements in that array. The value of the **accept_flags** parameter will usually be 0; other flag values currently defined for **exs_accept**() are discussed below. **qhandle** identifies an event queue on which a user can wait for the completion of this **exs_accept**().

The value returned in **result** will be 0 if the **exs_accept**() was successfully started, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno** and no asynchronous activity is started.

The array of structures pointed to by **address_vector** must have been allocated by the user before calling **exs_accept**(). Continuing the 1-800 analogy, there will be one element in this array for each agent available in the call center. Each element of the array contains the following three fields, which must be initialized by the user prior to calling **exs_accept**() as follows:

exs_addr
: is a pointer to a structure of type **struct sockaddr** into which the IPv4/IPv6 address and port number of a new remote client will be stored by the EXS interface (this is how the agent is able to identify the customer who is calling). If the user does not wish to get this information, this pointer can be NULL or the value of the **exs_addrlen** parameter can be 0.

exs_addrlen
: is the number of bytes allocated by the user to the structure pointed to by **exs_addr**. If that pointer is NULL, or if the user does not wish to get the remote client's IPv4/IPV6 address in the **exs_addr** parameter, the value of **exs_addrlen** can be 0.

exs_ahandle
: is an arbitrary pointer value chosen by the user for identification purposes only, and is never looked at by the EXS interface. It will be returned in the **exs_evt_ahandle** field of the event notification (as previously discussed in section 2.1.3).

Following the call to **exs_accept**(), the user must use:

nevents = exs_qdequeue(qhandle, event_vector, count, timeout);

to wait for clients to perform **exs_connect**() operations to this server's listening post. The **exs_qdequeue**() function has already been explained in section 2.1.3. To repeat from that section, **qhandle** must point to the same event queue as that used in the call to **exs_accept**(), **event_vector** must be a user provided array big enough to hold **count** events of type **exs_event_t**, and **timeout** is a pointer to a structure of type **struct timeval** containing the maximum amount of time the user wants to wait for an event to happen. The **timeout** parameter can be NULL if the user wants to wait indefinitely.

If the **nevents** value returned by **exs_qdequeue**() is positive, then each of the first **nevents** structures of the **event_vector** array were filled in by the EXS interface with information to identify each operation that posted an event to this event queue and to convey the final results of each operation back to the user. For an event associated with an **exs_accept**(), the following fields of the **exs_event_t** structure will be filled in (as generically described in section 2.1.3):

exs_evt_errno
: is 0 if the **exs_accept**() operation completed successfully, otherwise it contains an error code (i.e., a Linux **errno** value) to indicate why it failed.

exs_evt_socket
: is a copy of the listening post's **fd** value that was used as a parameter in the call to **exs_accept**(),

exs_evt_ahandle
: is a copy of the **ahandle** value stored by the user in the **exs_ahandle** field of an element in the array pointed to by the **address_vector** parameter to the **exs_accept().** This value is chosen by the user for identification purposes only, and is not used by the EXS interface.

exs_evt_type
: is **EXS_EVT_ACCEPT**.

exs_evt_union
: contains a structure of type **exs_evt_accept** that has been filled in by the EXS interface with values in the following fields:

  exs_evt_new_socket
  : is the file descriptor identifying the new connection to the client — this value should be used by the server from this point on to transmit data to/from that client. This is analogous in the 1-800 call center to the line to an exclusive agent to which the customer is switched. However, the analogy is not perfect because the call center line must already exist (as extension 123, for example), whereas this file descriptor represents a new socket created internally by the **exs_accept**() function. It is as if the **exs_accept**() internally calls **exs_socket**() for each new client, and the socket created by this internal call to **exs_socket**() has a file descriptor that is different from any existing file descriptors. Note that this scheme is not new to EXS — the **accept**() function for normal sockets works in exactly the same manner.

  exs_evt_addr
  : is a copy of the **exs_addr** pointer from an element in the array pointed to by the **address_vector** parameter to the **exs_accept**(). The structure pointed to by the **exs_evt_addr** field is of type **struct sockaddr**, and will now contain the IPv4/IPv6 address and port number of the remote client at the other end of the new connection.

  exs_evt_addrlen
  : contains the number of bytes used to store the client's IPv4/IPv6 address and port number in the structure pointed to by **exs_evt_addr**.

Once a new connection to a client has been indicated by a successful completion event, it is common for a server process to spawn a new "agent" thread to deal exclusively with that client. This is analogous to switching a customer's call to an agent in the 1-800 call center. The file

descriptor from the **exs_evt_new_socket** field in the **exs_evt_accept** structure should be used by this agent thread in all the **exs_send**() and **exs_recv**() functions for transactions with that client. When the agent thread's dealings with this client are finished, that thread should call **exs_close**() with this **fd** and then terminate. Just the agent thread needs to terminate, not the server listening post thread. Here the 1-800 analogy breaks down, because human agents will just hang up with one customer and wait for a call from another customer. With EXS, as with normal sockets, there is no way to reuse a socket — a new one must be created by **exs_accept**() as each new client connects.

The only **accept_flags** value currently defined for **exs_accept**() is **EXS_BLOCK**. When this flag value is present, the value of the **count** parameter MUST be exactly 1, since the **fd** for only 1 remote connection can be returned by 1 call. The value of the **qhandle** parameter is ignored by the EXS interface, and can be NULL. The **EXS_BLOCK** flag value should be supplied when the user wants the **exs_accept**() to operate synchronously rather than asynchronously. A synchronous **exs_accept**() blocks until a remote client connects, at which time the **result** returned by **exs_accept**() will be the **fd** for the new connection to the remote client. Note that in this case no **exs_event_t** structure is generated for the user, so the other fields in that structure are not available to the user. This means that the **exs_ahandle** field in the first (and only) **address_vector** is also ignored by the EXS interface, and can be NULL.

4.4. Closing an EXS connection

When the user has finished using a connection of any of the three types discussed above, the user calls the following function:

result = exs_close(fd, close_flags, qhandle, ahandle);

where **fd** identifies the connection to be closed, **flags** contains flags that modify the normal behavior of this call, **qhandle** identifies an event queue that will be used to wait for the completion of this call to **exs_close**(), and **ahandle** is an arbitrary pointer value chosen by the user which will be returned in the **exs_evt_ahandle** field of the event notification (discussed previously in section 2.1.3).

The value returned in **result** will be 0 if **exs_close**() was successfully started, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno** and no asynchronous activity is started.

If the **exs_close**() started successfully, it will operate asynchronously to the user thread that called it. During this time the user should not call any additional EXS functions using this **fd** to identify the connection, because the **fd** may become invalid and the internal state of the connection may become undefined once an **exs_close**() is called. When the **exs_close**() completes, the EXS interface will post to the user's **qhandle** an event whose **exs_evt_type** field contains the value **EXS_EVT_CLOSE**. The **exs_evt_errno**, **exs_evt_socket**, and **exs_evt_ahandle** fields of the event will be filled in as previously described in section 2.1.3. No additional information is stored in the **exs_evt_union** field for events of this type.

It is highly recommended that, prior to exiting a program or thread, a user always call **exs_close**() for each connection controlled by that program or thread. The reason for this is that EXS is inherently asynchronous, so many EXS functions called by a user simply start an asynchronous operation — the real work takes place asynchronously to the thread that called the EXS

function. Therefore, when writing code, a user might not be aware of all that is being accomplished asynchronously, so that what looks to be finished to a user might not be finished in the EXS interface. Calling the **exs_close**() function ensures that this asynchronous activity is finished, whereas exiting the program or thread would not ensure this, and thus all data might not be transmitted. Calling **exs_close**() also helps to cleanly shutdown the other end of the connection.

The only **close_flags** value currently defined for **exs_close**() is **EXS_BLOCK**. When this flag value is present, the value of the **qhandle** and **ahandle** parameters are ignored by the EXS interface, and can be NULL. The **EXS_BLOCK** flag value should be supplied when the user wants the **exs_close**() to operate synchronously rather than asynchronously.

5. Basic data transfer over an EXS connection

There are two basic, complementary operations that transfer data over an EXS connection: **exs_send**() and **exs_recv**(). These two operations are used regardless of how the connection was established, since both clients and servers need to be able to both send and receive data. Both of these functions are asynchronous, so they only start the data transfer — the user must explicitly call **exs_qdequeue**() to know when the transfer finishes.

5.1. Sending data asynchronously

A user sends data asynchronously by using:

result = exs_send(fd, send_buffer, send_length, send_flags, qhandle, ahandle, mhandle);

where **fd** identifies the connection, and the user has filled **send_buffer** with **send_length** bytes of data prior to calling **exs_send**(). This **send_buffer** must be completely within an area of memory that was assigned the memory registration key **mhandle** in a previous call to **exs_mregister**(). **send_flags** will usually have the value 0; other flags currently defined for **exs_send**() are discussed in sections 9 and 10 below. **qhandle** identifies an event queue previously created by the user to wait for completion events. **ahandle** is an arbitrary pointer value chosen by the user which will be returned in the **exs_evt_ahandle** field of the completion event notification (discussed previously in section 2.1.3).

5.2. Receiving data asynchronously

A user receives data by using:

result = exs_recv(fd, recv_buffer, max_length, recv_flags, qhandle, ahandle, mhandle);

where **fd** identifies the connection, and the user has reserved a **recv_buffer** capable of holding **max_length** bytes of data. This **recv_buffer** must be completely within an area of memory that was assigned the memory registration key **mhandle** in a previous call to **exs_mregister**(). **recv_flags** will usually have the value 0, unless other flags are desired. As of version 1.3.0, for **SOCK_STREAM** sockets, the **MSG_WAITALL** flag will make the **exs_recv**() operation wait until it can fill all **max_length** bytes; otherwise, the operation will complete as soon as any data is received. For **SOCK_SEQPACKET** sockets, the **MSG_WAITALL** flag has no effect. Other effects of this flag and additional flags currently defined for **exs_recv**() are discussed in sections 9 and 10 below. **qhandle** identifies an event queue previously created by the user to wait for completion events. **ahandle** is an arbitrary pointer value chosen by the user which will be

returned in the **exs_evt_ahandle** field of the completion event notification (discussed previously in section 2.1.3).

5.3. Waiting for asynchronous I/O completion events

The value returned by **exs_send**() or **exs_recv**() in **result** will be 0 if the operation was successfully started, or -1 if there was an error of some sort, in which case the error code is stored in the global **errno** and no asynchronous activity is started.

Once an **exs_send**() or **exs_recv**() is successfully started, it will operate asynchronously to the user program. During this time the user must not in any way modify the area of memory pointed to by the buffer parameter, because the data in that buffer has still not been transferred across the connection. Eventually the user must use:

nevents = exs_qdequeue(qhandle, event_vector, count, timeout);

to wait for the operation to finish so that the buffer can be safely accessed again. The **exs_qdequeue**() call was explained in section 2.1.3. **qhandle** must be the value used in the **exs_send**() or **exs_recv**(), **event_vector** must be an array big enough to hold **count** events of type **exs_event_t**, and **timeout** is the maximum amount of time the user wants to wait for an event to happen (NULL for an indefinite wait).

If the value returned in **nevents** is positive, then each of the first **nevents** structures of the **event_vector** array were filled in by the EXS interface with information to identify the operation that caused this event and to convey the final results of that operation back to the user. For each of these structures, the fields **exs_evt_errno**, **exs_evt_socket**, and **exs_evt_ahandle** will be filled in by the EXS interface with the values explained previously (in section 2.1.3). In addition:

exs_evt_type
: will indicate which type of operation caused the event: **EXS_EVT_SEND** if the operation was **exs_send**(), or **EXS_EVT_RECV** if the operation was **exs_recv**().

exs_evt_union
: will contain a structure of type **exs_evt_xfer** that contains the following fields:
  — **exs_evt_buffer** is the buffer address specified in the **exs_send**() or **exs_recv**()
  — **exs_evt_mhandle** is the **mhandle** specified in the **exs_send**() or **exs_recv**()
  — **exs_evt_amount_lost** is described below in section 5.4
  — **exs_evt_length** is the number of bytes successfully transferred by this operation.
  When the value of **exs_evt_type** is **EXS_EVT_SEND** and the operation was successful, the value of **exs_evt_length** will always be equal to the value of the **send_length** parameter in the original **exs_send**(). When the value of **exs_evt_type** is **EXS_EVT_RECV** and the operation was successful, the value of **exs_evt_length** will always be less than or equal to the value of the **max_length** parameter in the original **exs_recv**().

5.4. Matching sends with receives

Since the amount of data sent in one packet by an **exs_send**() may differ from the amount of data requested in the remote side's corresponding **exs_recv**(), the EXS interface layer must rationalize any difference. For efficiency, the way it does this is very packet-oriented, and hence differs somewhat from the way this is done in traditional sockets.

5.4.1. Receiver's buffer is greater than or equal to amount of data in sender's packet

If the receiver provides a buffer whose size is greater than or equal to the size of the data in the matching send packet, there is no problem — the EXS interface delivers all the sent data into the beginning of the receiver's buffer, any remaining space at the end of the receiver's buffer is left undefined, the value returned in **exs_evt_length** is the exact number of bytes delivered into the buffer, and **exs_evt_amount_lost** is always zero.

5.4.2. Receiver's buffer is less than amount of data in sender's packet

If the receiver provides a buffer whose size is less than the size of the data in the matching send packet, the resolution depends on the type of socket in use. For all socket types, the EXS interface completely fills the receiver's buffer with the first part of the sent data, and returns in **exs_evt_length** the exact number of bytes delivered into the buffer.

If the socket type is **SOCK_STREAM**, the EXS interface ignores packet boundaries, so it saves the remainder of the data that is in the matching send packet and uses it to match with subsequent **exs_recv**() calls. No data is ever lost, so the value returned to the receiver in **exs_evt_amount_lost** is always zero.

If the socket_type is **SOCK_SEQPACKET**, the EXS interface maintains packet boundaries, so it discards the remainder of the data in the matching send packet, and returns to the receiver in **exs_evt_amount_lost** the number of bytes discarded.

6. Basic flow control within the EXS interface

Because data sent by **exs_send**() operations on one end of a connection must be delivered into buffers specified by **exs_recv**() operations on the other end, the EXS interfaces on both ends must coordinate the flow of this data so that neither side runs out of buffers. This is done in a manner that is largely transparent to the user.

Since a user must allocate and fill a memory buffer before specifying it as the **send_buffer** parameter to **exs_send**(), the EXS interface does not have to provide any additional storage for data on the sending side. However, the interface cannot actually send data until it knows for sure that the user on the receiving side has provided a corresponding buffer into which the sender's data can be delivered without any additional copying or buffering (because EXS uses direct memory to memory transfers). The EXS interface uses a credit mechanism to accomplish this.

6.1. Send and Receive Credits

The interface on each end of an established connection internally maintains two dynamically varying local credit values. At any time "send_credits" is the maximum number of packets this interface is allowed to start sending to the other side using **exs_send**(); and "recv_credits" is the maximum number of packets this interface is allowed to start receiving from the other side using **exs_recv**(). At any time the value of **send_credits** on one side must equal the value of **recv_credits** on the other side, and vice versa. As explained in the next section (6.2), when an EXS connection is first established, the EXS interfaces on both sides negotiate these numbers so that they are initialized to the same values.

Each time a user issues an **exs_recv**(), the receiving EXS interface reduces its local **recv_credits** by one. If the balance would become negative, the receiving interface blocks the

exs_recv() operation (i.e., it will not proceed) until more local **recv_credits** become available (as explained below). (However, see the **EXS_DONTWAIT** flag described in section 10.) If the balance would not become negative, the receiving interface sends an "advertisement" to the sending side and then returns a value of 0 to the caller of the **exs_recv**() to indicate that the **exs_recv**() has started successfully. This advertisement contains no data, but rather information (or "metadata") describing the memory on the receiver that is now ready to receive data from the sender (i.e., its length, location, and memory registration key).

The sending interface must keep track locally of the **send_credits** it has negotiated with the receiver. Each time a sending user issues an **exs_send**(), the sending EXS interface reduces its local **send_credits** by one. If the balance would become negative, the sending interface blocks the **exs_send**() operation (i.e., it will not proceed) until more local **send_credits** become available (as explained below). (However, see the **EXS_DONTWAIT** flag described in section 10.) If the balance would not become negative, the sending interface adds the information (i.e., "metadata") from this **exs_send**() to an internal queue and returns a value of 0 to the caller of the **exs_send**() to indicate that the **exs_send**() has been started successfully.

6.2. Negotiations at connection establishment

At the time an EXS connection is first established, the client side automatically sends the server a short "setup request" message that contains the client's EXS version number (currently 1), and the initial values of the client's **send_credits**, and **recv_credits**. Upon receiving this "setup request" message, the server side of the connection compares the values contained in this message with its own corresponding values. The **minimum** of each corresponding value is used to reset the server's own value and to build a "setup response" message that it sends back to the client. Once the client receives this "setup response" message, it uses those values to set its own corresponding values. Consequently, from that point in time on, both ends of the newly established connection have the same value for each of the corresponding parameters, and the flow control mechanism will now function properly. These negotiations are transparent to the user.

Following these negotiations, the EXS interface on each side of a newly established connection sets up one internal receive buffer for each local **send_credit** that it has negotiated with the corresponding receiving side. These buffers are used to store EXS advertisements as they arrive from the remote end. When the receiving interface sends an advertisement to the sender, the advertisement is delivered directly into one of these interface buffers and the sending interface is notified of this arrival by the OFED stack. Since the receiving interface should never send more advertisements than it has local **recv_credits**, and since the sending interface has posted one receive buffer for each of its local **send_credits**, no advertisements should ever be lost for lack of a buffer when they arrive on the sending side.

6.3. Matching advertisements and receives

Each EXS interface keeps track of advertisements it has received and **exs_send**() operations that its sending user has started. Whenever the user starts an **exs_send()** operation, the EXS interface looks for an already received advertisement to match it with. Similarly, whenever an advertisement is received from the remote receiver, the receiving interface looks for an already started **exs_send()** to match it with. Matching occurs in the manner already described in section 5.4. The sending interface then issues an RDMA_WRITE_WITH_IMM operation to its local

CA (via the OFED stack) in order to actually transfer data into the user's buffer on the receiving side (as indicated in the advertisement) directly from the user's buffer on the sending side (as indicated in the **exs_send**()) without any extra copying or CPU intervention on either side. This RDMA_WRITE_WITH_IMM operation is transparent to the user on both sides of a connection.

When the receiving interface is notified by its receiving CA (via the OFED stack) that a remotely issued RDMA_WRITE_WITH_IMM operation has completed, it will do two things.

(1) It uses the "immediate" value supplied in the local RDMA_WRITE_WITH_IMM completion to locate the corresponding advertisement previously issued by that receiver and posts the receiver's completion event to tell the receiving user that its **exs_recv**() has completed, and to convey the results to the receiving user via the **exs_event_t** structure.

(2) It increments it's local **recv_credits**, and if other **exs_recv**() operations were blocked waiting for a credit, the next one of these in sequence is allowed to proceed.

When the sending interface is notified by its sending CA (via the OFED stack) that the locally issued RDMA_WRITE_WITH_IMM operation has completed, it also does two things.

(1) It posts the sender's completion event to tell the sending user that its **exs_send**() has completed and to convey the results to the sending user via the **exs_event_t** structure.

(2) It increments its local **send_credits** for this connection and if other **exs_send**() operations were blocked waiting for a credit, the next one of these in sequence is allowed to proceed.

Clearly this credit mechanism requires each EXS interface to allocate some hidden, internal buffers for exchanging advertisements. However, these buffers and advertisements are small, since they are used only to send and receive a limited amount of control information, not an unlimited amount of user data. Indeed, these buffers are small enough to be embedded in internal control blocks that contain additional information needed locally by the EXS interface.

6.4. Number of internal buffers allocated

Using the mechanism discussed in the next section, a user can set the local values to use when negotiating the initial **send_credits** and **recv_credits** values. Therefore, it is important for the user to understand how the EXS interface uses these values to allocate its internal buffers. The receiving interface needs two control blocks for each local **recv_credit** it has initially negotiated with the sending interface: one for sending an advertisement, and one for receiving the corresponding remotely issued RDMA_WRITE_WITH_IMM completion. The sending interface also needs two control blocks for each local **send_credit** that it has negotiated with the receiver: one for keeping track of waiting **exs_send**() operations that have been started but for which no advertisement has been received yet, and one for receiving an advertisement. However, one extra control block is allocated for each local **send_credit** to ensure that receive buffers are always posted regardless of the inevitable time delays between sending or receiving a message and being notified of the completion of that operation (at which time a control block can be reused).

7. Tuning the UNH EXS interface

One of the differences between the UNH EXS interface and the ES-API is an additional mechanism by which a user can "tune" some aspects of the EXS interface in order to increase

performance for a particular application. The user does this through the use of:

result = exs_fcntl(fd, command, argument);

The value of the **fd** parameter identifies the socket to be tuned, the value of the **command** parameter indicates what the user wants to do to the socket, and the value of the **argument** parameter depends on the **command**. This is a synchronous operation, so there is no event associated with its completion. The value returned in **result** will also depend on the command, although for all commands it will be -1 if there was an error of some sort, in which case the error code is stored in the global **errno**.

7.1. Credit negotiation

As previously discussed in section 6, the credit value used to control the flow of data in EXS is negotiated at the time a new connection is established. The value used in this negotiation can be set by the user through the use of the **exs_fcntl**() function prior to the **exs_connect**() and **exs_accept**() calls on the **fd** parameter. When the value of the **command** parameter to the **exs_fcntl**() function is **EXS_F_SETFLOWCONTROLCREDITS**, the EXS interface will use the value of the **argument** parameter as the local value to be used in the negotiation of the local **send_credits** and **recv_credits**. The default value is 32. In previous versions, 32 was also the maximum acceptable value, but this maximum has been removed as of version 1.3.0. Negotiation occurs as part of connection establishment, and the result is the minimum value supplied by either side. This **command** parameter value cannot be used in an **exs_fcntl**() call on an established connection. The result returned by a successful **exs_fcntl**() call is the old value of the corresponding local credit value.

When the value of the **command** parameter to the **exs_fcntl**() function is **EXS_F_GETFLOWCONTROLCREDITS**, the result returned by the **exs_fcntl**() depends on whether or not the connection has been established. If **exs_fcntl**() is called with this **command** parameter value prior to a successful call to **exs_connect**() or **exs_accept**() on the **fd** parameter, the value returned will be the local value used in a future negotiation on that connection. If it is called after a connection was successfully established, the value returned will be the local value that resulted from the negotiation at the time the connection was established. Note that this return value is not the dynamically varying credit value used to control the flow on the connection, but the negotiated limit on that credit value.

7.2. Small unregistered packets

Because of the overhead involved in dynamically registering and unregistering memory (see section 8), it may be faster to send and receive small amounts of data by having the EXS library simply copy the data into/out of preregistered library buffers. Thus, in **SOCK_SEQPACKET** mode, UNH EXS supports copying unregistered small packets. The definition of a “small” packet can be controlled through the use of the **EXS_F_SETSPMAXSIZE** value of the **command** parameter to the **exs_fcntl**() function, in which case the value of the **argument** parameter should be the value to be used in the negotiation of the definition of “small”. The default value is 0. Negotiation occurs as part of connection establishment, and the result is the minimum value supplied by either side. This **command** parameter value cannot be used in an **exs_fcntl**() call on an established connection. The result returned by a successful **exs_fcntl**() call is the old value of the corresponding local small packet max size.

Setting a positive value for **EXS_F_SETSPMAXSIZE** effects the operation of **exs_send**() and **exs_recv**() called with a value of **EXS_MHANDLE_UNREGISTERED** for the required **mhandle** parameter. If the value of the required **length** parameter is less than or equal to the small packet max size, then rather than dynamically registering and unregistering the data supplied in the function call, the data is copied into a preregistered library buffer on **exs_send**() or copied out of a preregistered library buffer on **exs_recv**().

When the value of the **command** parameter to the **exs_fcntl**() function is **EXS_F_GETSPMAXSIZE**, the result returned by the **exs_fcntl**() depends on whether or not the connection has been established. If **exs_fcntl**() is called with this **command** parameter value prior to a successful call to **exs_connect**() or **exs_accept**() on the **fd** parameter, the value returned will be the local value used in a future negotiation on that connection. If it is called after a connection was successfully established, the value returned will be the local value that resulted from the negotiation at the time the connection was established.

In **SOCK_STREAM** mode, this is not supported since the stream mode does its own buffering by default.

7.3. Using the hardware inline feature on exs_send()

Many RDMA interface cards support the optional "inline" feature, which allows the card to enqueue a copy of small amounts of data as part of the metadata it enqueues on the send queue at the time it starts an **exs_send**() operation. The effect of this is to make the transfer slightly faster (i.e., to exhibit lower latency), because the data is already on the interface card at the time it actually moves data onto the wire, so there is no need to involve the memory bus in the transfer itself (the memory bus was involved at the time the **exs_send**() was enqueued).

The definition of "small amounts" of data can be controlled to some extent through the use of the **EXS_F_SETINLINEMAXSIZE** value of the **command** parameter to the **exs_fcntl**() function, in which case the value of the **argument** parameter should be the definition of "small amounts". The default value is the largest value acceptable to the interface card. Negotiation occurs as part of connection establishment, and is explained next. This **command** parameter value cannot be used in an **exs_fcntl**() call on an established connection. The result returned by a successful **exs_fcntl**() call is the old value of the corresponding local inline max size.

The user does not have complete control over this value, because this is an optional feature of RDMA interface cards, and the maximum size possible depends on the particular card used. Therefore, if the user uses the **EXS_F_SETINLINEMAXSIZE** value in the **command** parameter to the **exs_fcntl**() function, then the user's value of the **argument** parameter is only a starting point for the EXS library to negotiate with the local interface card to set the maximum inline size to use. If the user's value is acceptable to the local interface card, then that is the value used. If the user's value is not acceptable, then the EXS library will silently find and use the largest value acceptable to the interface card but smaller than the user's value.

Note that the scope of the maximum inline size is the local interface card — it is not negotiated with the remote side when a connection is established by an **exs_connect**() or **exs_accept**(), because the interface card on the remote side could be different from the local interface card, and might support a different maximum value. In any case, maximum inline size is only utilized by the EXS library on an **exs_send**(), since it only effects the queueing performed locally when this

function is called — it has no effect on an **exs_recv**() or on data transferred on the wire.

Note also that the maximum inline size is independent of the small packet max size. The maximum inline size is a hardware option that improves the latency of **exs_send**() operations, regardless of whether or not the **mhandle** parameter is given as **EXS_MHANDLE_UNREGISTERED** in the **exs_send**(). Therefore the EXS library will use it by default on all calls to **exs_send**() in which the value of the **send_length** parameter is less than or equal to the maximum inline size. (Obviously if the user sets this maximum to 0, this feature will not be used.) The small packet max size is a software option that improves the latency of **exs_send**(), but only when the **mhandle** parameter is given as **EXS_MHANDLE_UNREGISTERED**. The EXS library uses will use it only on these calls, and only if the value of the **send_length** parameter is less than or equal to the small packet max size.

When the value of the **command** parameter to the **exs_fcntl**() function is **EXS_F_GETINLINEMAXSIZE**, the result returned by the **exs_fcntl**() depends on whether or not the connection has been established. If **exs_fcntl**() is called with this **command** parameter value prior to a successful call to **exs_connect**() or **exs_accept**() on the **fd** parameter, the value returned will be the local value used in a future negotiation with the local interface when a connection is established. If it is called after a connection was successfully established, the value returned will be the local value that resulted from the negotiation at the time the connection was established.

7.4. Pinning the EXS completion thread to a CPU

Because of the asynchronous nature of its operation, the EXS library utilizes a “completion thread”. One of the fine-tuning options available to users is the ability to pin this completion thread to a particular CPU. If the user pins his/her threads to different CPUs, the execution load will be distributed across a multi-core platform, which should give better performance. If thread pinning is not performed, then the completion thread can be dynamically assigned to various CPUs by the kernel scheduler.

Completion thread pinning is controlled through the use of the **EXS_F_SETCOMPTHREADCPU** value of the **command** parameter to the **exs_fcntl**() function, in which case the value of the **argument** parameter should be the number (starting at 0) of the CPU to which the completion thread for the connection indicated by the **fd** parameter should be pinned. The default value is INT_MAX, which means that the completion thread is NOT pinned to any CPU. Pinning is done at the time a connection is established, and may be changed dynamically after connection establishment. The result returned by a successful **exs_fcntl**() call is the old CPU number. If this return value is INT_MAX, it means the completion thread was not previously pinned to any CPU, and can be dynamically assigned to CPUs by the kernel scheduler.

7.5. Busy polling for completions

By default, the UNH EXS completion thread releases the CPU whenever it has no work to do in order to not consume CPU cycles unnecessarily. However, this requires kernel intervention, both to release the CPU and again when the thread needs to be reawakened, and this can add several microseconds overhead to the performance of an EXS transaction. To avoid this, the user can select the “busy polling” option for a completion thread. This option means that the completion thread will react faster to the completion of transactions, which usually produces

lower latency. But it also means the completion thread will never give up the CPU, so that it will consume 100% of the available cycles on a CPU.

Busy polling in the completion thread of the connection indicated by the **fd** is controlled through the use of the **EXS_F_SETFD** value of the **command** parameter to the **exs_fcntl**() function, in which case the value of the **argument** parameter should be a bit-mask containing the **EXS_FD_BUSYPOLL** flag. The choice used by a completion thread is determined when that thread is created as part of connection establishment. By default, this flag is not set, which means that busy polling is not employed by the completion thread for an **fd**. The **argument** bit-mask containing the **EXS_FD_BUSYPOLL** flag cannot be used in an **exs_fcntl**() call on an established connection. The result returned by a successful **exs_fcntl**() call is the old flags bit-mask.

When the value of the **command** parameter to the **exs_fcntl**() function is **EXS_F_GETFD**, the result returned by the **exs_fcntl**() is the current flags value at that time. The value of the **argument** parameter is ignored.

The proper way to set the **EXS_FD_BUSYPOLL** flag is to:

call **exs_fcntl**() with the **EXS_F_GETFD** value for the **command** parameter,

OR the result returned by that **exs_fcntl**() call with the **EXS_FD_BUSYPOLL** flag,

use that result of that OR operation as the **argument** parameter in a call to **exs_fcntl**() with the **EXS_F_SETFD** value for the **command** parameter.

Doing it this way ensures that only a single flag value (in this case, the **EXS_FD_BUSYPOLL** flag) gets changed.

7.6. Stream receive buffer

As mentioned earlier in section 4.1, version 1.3.0 introduces an intermediate receive buffer for **SOCK_STREAM** sockets. This buffer is meant to decrease latency for long-distance communications that is introduced by the advertisement mechanism. When no advertisements are pending, the sender will write data into this intermediate buffer instead of waiting for an advertisement. **exs_recv**() will return data from this buffer whenever it contains data, and will send advertisements only if the **MSG_WAITALL** flag is set and there is no data in the buffer. The sender will make a best effort to write directly to the user buffer whenever it gets an advertisement. However, it is possible that the sender had already written the data to the intermediate buffer before it received the advertisement, in which case it will ignore the advertisement.

The intermediate receive buffer defaults to 3 Gigabits in size, and this is also its maximum size due to limitations of the underlying OFED stack. There is a fallback mechanism to decrease the size if the requested size cannot be allocated. However, due to the way that memory allocation works in Linux and the fact that this intermediate buffer must be pinned in virtual memory, this fallback mechanism is unlikely to work in practice if the available physical memory is less than the desired buffer size.

To decrease the size of the intermediate receive buffer, call **exs_fcntl**() with the **command** parameter set to **EXS_F_SETSTREAMBUFSIZE** and the **argument** parameter set to the

desired size in bytes. This parameter must be set on the client side of the connection before calling **exs_connect**() and on the server side before calling **exs_accept()** for the first time. The similar **EXS_F_GETSTREAMBUFSIZE** may be used at any time to determine the actual size of the intermediate receive buffer, which may be less than the requested size due to memory constraints.

In many cases, the intermediate receive buffer will increase performance dramatically; however, there may be some use cases where this extra buffer is unwanted. To turn this feature off completely, use the **EXS_FD_NODELAY** flag, which is applied in the same way as the busy polling flag described above.

In the future, we plan to additionally support a send-side Nagle buffer for small packets. The size of this buffer would then be controllable via the **EXS_F_SETSPMAXSIZE** command to the **exs_fcntl**() function mentioned in the previous section.

8. Registered and unregistered memory

The EXS interface is designed to transfer data using direct memory-to-memory transfers with no extra copying. This requires that memory buffers involved at each end of the RDMA transmission be "registered" with the CA on each end prior to one side issuing the **exs_send**() and the other side issuing the **exs_recv**(). This normally requires the user to explicitly register and deregister the memory used in EXS transfers, on both the sending and receiving sides (see section 8.1). However, it is possible for a user to implicitly register and deregister the memory used on either the sending or receiving side of an EXS transfer, or both (see section 8.2).

8.1. Explicit memory registration and deregistration

A user explicitly registers memory using:

mhandle = exs_mregister(address, length, flags);

where **address** points to an area of memory containing **length** bytes to be registered. The value of **flags** is usually **EXS_ACCESS_ALL** to grant read and write access to both the local and remote ends of a connection. Note that prior to calling **exs_mregister**(), the memory at **address** must already be allocated by the user, either statically or dynamically.

If the exs_mregister() is successful, the value returned in **mhandle** will be an opaque memory handle that can be used as a parameter to an **exs_send**() when the send buffer is located anywhere within this area of memory, and/or as a parameter to an **exs_recv**() when the receive buffer is located anywhere within this area of memory. If the **exs_mregister**() fails for any reason, the value returned in **mhandle** will be the constant EXS_MHANDLE_INVALID and an error code will be stored in the global **errno**.

Note that once an **mhandle** has been successfully registered, it can be used repeatedly in subsequent calls to **exs_send**() and/or **exs_recv**() that have buffers in that memory area. Only when a user is completely finished using an explicitly registered area of memory for I/O does he or she deregister it using:

result = exs_mderegister(mhandle, flags);

where **mhandle** must be the value returned by a previously successful call to **exs_mregister**(), and the value of **flags** must be 0 since no flags are currently supported for **exs_mderegister**().

8.2. Implicit memory registration and deregistration:

The EXS standard provides a simple mechanism to implicitly register and deregister memory buffers used in **exs_send**() and **exs_recv**() operations. To do this, wherever an **mhandle** parameter is required in an EXS function call, the user simply supplies the constant **EXS_MHANDLE_UNREGISTERED** to indicate that the user has not explicitly registered the buffer parameter specified in that function call. Given this value, the EXS interface will dynamically register and deregister the buffer as necessary. Of course this adds considerable overhead interface to dynamically register and deregister memory as part of an **exs_send**() or **exs_recv**() call, but if an area of memory is used for I/O only once or twice, rather than repeatedly, the user may find it more convenient to let the EXS interface perform the registration required by the CA rather than coding out explicit calls to **exs_mregister**() and **exs_mderegister**().

Note that memory registration applies only to the process calling the **exs_mregister**() function (and to threads attached to that process). If a process forks a child process, that child does not inherit any of the parent's memory registrations.

9. Synchronous I/O

The EXS interface is designed to transfer data asynchronously, as already described in section 5. However, some applications have no need for asynchronous I/O, although they still want to use RDMA. To accommodate this type of application, the user can simply supply the value **EXS_BLOCK** in the **flags** parameter to **exs_accept**(), **exs_connect**(), **exs_close**(), **exs_send**(), or **exs_recv**(). When this flag is present, **qhandle** and **ahandle** parameters required in these EXS function calls are ignored by the EXS interface and can be NULL. The **EXS_BLOCK** flag indicates that the user wants this function to both start an operation and wait for its completion. If the result returned by this function is -1, the error code may apply either to the start phase or the completion phase — the user has to somehow determine which. Otherwise, the result is a value taken from one of the fields in the successful completion event (which is hidden from the user by the EXS interface). For an **exs_send**() or **exs_recv**(), the result will be the number of bytes actually transmitted. For an **exs_accept**() it will be the **fd** of the new connection. For an **exs_connect**() or **exs_close**() it will be 0.

For convenience when the user wishes to both block and use unregistered memory in a send or receive operation, two additional functions have been provided: **exs_write**() and **exs_read**(). These are described below in terms of their **exs_send**() and **exs_recv**() equivalents.

9.1. Sending data synchronously

A user sends registered data synchronously by using:

result = exs_send(fd, write_buffer, write_length, EXS_BLOCK, NULL, NULL, mhandle);

or the more convenient:

result = exs_blocking_send(fd, write_buffer, write_length, send_flags, mhandle);

in which the user does not have to supply the value **EXS_BLOCK** in the **send_flags** parameter.

A user sends unregistered data synchronously by using either:

result = exs_send(fd, write_buffer, write_length, EXS_BLOCK, NULL, NULL,

EXS_MHANDLE_UNREGISTERED);

or the more convenient:

result = exs_write(fd, write_buffer, write_length);

In all these situations, **fd** identifies the EXS connection, and the user has filled **write_buffer** with **write_length** bytes of data prior to the call. When using these functions, the EXS interface will automatically wait for completion of the data transfer before returning. If the transfer is successful, the value returned in **result** will be the number of bytes actually written. Otherwise, the value returned in **result** will be -1 to indicate that there was an error of some sort, in which case the error code is stored in the global **errno**.

9.2. Receiving data synchronously

A user receives registered data synchronously by using:

result = exs_recv(fd, read_buffer, max_length, EXS_BLOCK, NULL, NULL, mhandle);

or the more convenient:

result = exs_blocking_recv(fd, read_buffer, max_length, recv_flags, mhandle);

in which the user does not have to supply the value **EXS_BLOCK** in the **recv_flags** parameter.

A user receives unregistered data synchronously by using either:

result = exs_recv(fd, read_buffer, max_length, EXS_BLOCK, NULL, NULL, EXS_MHANDLE_UNREGISTERED);

or the more convenient:

result = exs_read(fd, read_buffer, max_length);

In all these situations, **fd** identifies the EXS connection, and the user has reserved a **read_buffer** capable of holding **max_length** bytes of data. When using these functions, the EXS interface will automatically wait for completion of the data transfer before returning. If the transfer is successful, the value returned in **result** will be the number of bytes actually read. Otherwise, the value returned in **result** will be -1 to indicate that there was an error of some sort, in which case the error code is stored in the global **errno**.

9.3. Establishing an EXS client connection synchronously

The **EXS_BLOCK** constant can also be used as the value of the **flags** parameter in the the **exs_connect**() function so that it will operate synchronously rather than asynchronously. When this flag value is present, the value of the **qhandle** and **ahandle** parameters to this function are ignored by the EXS interface, and can be NULL. For convenience and similarity to the equivalent functions for normal sockets, the following synchronous function is also provided:

result = exs_blocking_connect(fd, server_address, server_addrlen);

where the parameters are identical to those for the corresponding “normal” TCP/IP socket **connect**() function.

9.4. Accepting connections on the server synchronously

The **EXS_BLOCK** constant can also be used as the value of the **flags** parameter in the the **exs_accept**() function so that it will operate synchronously rather than asynchronously. When this flag value is present, the value of the **qhandle** parameter to this function is ignored by the EXS interface, and can be NULL. For convenience and similarity to the equivalent function for normal sockets, the following synchronous function is also provided:

result = exs_blocking_accept(fd, peer_address, &peer_addrlen);

exs_blocking_accept() blocks until a remote client connects, at which time the **result** it returns will be the **fd** for the new connection to the remote client. No **exs_event_t** structure is generated for the user, so the other fields in that structure are not available to the user.

It is important to note that the second and third parameters to **exs_blocking_accept**() differ from the corresponding parameters to **exs_accept**() (see section 4.3.2). Instead of giving an **address_vector** array of structures of type **exs_acceptaddr** and a **count** parameter indicating the number of elements in the array, **exs_blocking_accept**() takes as its **peer_address** parameter a pointer to a structure of type **struct sockaddr** into which the IPv4/IPv6 address and port number of a new remote client will be stored by the EXS interface. The **peer_addrlen** parameter is the number of bytes allocated by the user to the structure pointed to by **peer_address**. These two parameters to **exs_blocking_accept**() are identical to the first two fields in an element of the **address_vector** parameter to **exs_accept**(), and make the 3 parameters to **exs_blocking_accept**() identical to the 3 parameters of the "normal" TCP/IP socket function **accept**().

Note also that the **EXS_BLOCK** flag does NOT have to be included in the **accept_flags** parameter to **exs_blocking_accept**().

9.5. Closing EXS connections synchronously

An EXS connection of any type can be closed synchronously by using either:

result = exs_close(fd, EXS_BLOCK, NULL, NULL);

or the more convenient:

result = exs_blocking_close(fd);

where **fd** indicates the connection to be closed.

10. Converting programs from using normal sockets to using EXS sockets

The availability in EXS of both implicit memory registration and synchronous I/O means that users who wish to convert existing programs using "normal" sockets (see section 10.1) to full use of EXS sockets have a choice of how to proceed. They can start by using EXS sockets in synchronous mode and use only implicit memory registration (see section 10.2). Such usage is almost identical with the use of "normal" sockets (essentially just the function names change), but it does give the user access to the RDMA hardware. Once that is working, the user can choose between converting first to using explicit memory registration while continuing to use synchronous mode (see section 10.3), or converting first to using asynchronous mode while continuing to use implicit memory registration (see section 10.4). Once that step has been finished and is working properly, the other step can be taken to give fully asynchronous operation with explicitly registered memory (see section 10.5).

10.1. Client using normal sockets

The following gives an example of the complete conversion just mentioned, starting with the general outline for a client that uses “normal” sockets (and therefore cannot use RDMA hardware):

```
fd = socket(PF_INET, SOCK_STREAM, 0);
connect(fd, server_address, server_addrlen);
loop
        write(fd, out_buffer, out_bytes);
        in_bytes = read(fd, in_buffer, maxbytes);
endloop;
close(fd);
```

10.2. Client using EXS sockets in synchronous mode with implicit memory registration

The first conversion step mentioned above gives an almost identical program that uses synchronous mode and implicit memory registration so that it can therefore use RDMA hardware. Note that this program is identical to the normal program except for the introduction of the one-time call to **exs_init**() at the start, and the name changes of the various socket functions.

```
exs_init(EXS_VERSION);
fd = exs_socket(PF_INET, SOCK_STREAM, 0);
exs_blocking_connect(fd, server_address, server_addrlen);
loop
        exs_write(fd, out_buffer, out_bytes);
        in_bytes = exs_read(fd, in_buffer, maxbytes);
endloop;
exs_blocking_close(fd);
```

10.3. Client using EXS sockets in synchronous mode with explicit memory registration

We now have a choice of which feature of EXS to apply first. Let's choose to register memory before we go to asynchronous operation (so we will continue to use only EXS functions that are "blocking"). The general outline for this version of the client would be:

```
exs_init(EXS_VERSION);
fd = exs_socket(PF_INET, SOCK_STREAM, 0);
exs_blocking_connect(fd, server_address, server_addrlen);
in_mhandle = exs_mregister(in_buffer, max_in_bytes, flags);
out_mhandle = exs_mregister(out_buffer, max_out_bytes, flags);
loop
        exs_blocking_send(fd, out_buffer, out_bytes, 0, out_mhandle);
        in_bytes = exs_blocking_recv(fd, in_buffer, in_bytes, 0, in_mhandle);
endloop;
exs_mderegister(out_mhandle);
exs_mderegister(in_mhandle);
exs_blocking_close(fd);
```

10.4. Client using EXS sockets in asynchronous mode with implicit memory registration

Alternatively, we could choose to utilize EXS asynchronous operations before registering memory. The general outline for this version of the client would be:

```
exs_init(EXS_VERSION);
fd = exs_socket(PF_INET, SOCK_STREAM, 0);
management_qhandle = exs_qcreate(1);
exs_connect(fd, server_address, server_addrlen, 0, NULL, management_qhandle,
                                    NULL);
/*---- perform computation in parallel with EXS activity ----*/
exs_qdequeue(management_qhandle, &management_event, 1, NULL);
in_qhandle = exs_qcreate(3);
out_qhandle = exs_qcreate(3);
loop
        exs_send(fd, out_buffer, out_bytes, 0, out_qhandle,
                                        NULL, EXS_MHANDLE_UNREGISTERED);
        /*---- perform computation in parallel with data transfer ----*/
        exs_qdequeue(out_qhandle, &out_event, 1, NULL);
        exs_recv(fd, in_buffer, max_in_bytes, 0, in_qhandle,
                                        NULL, EXS_MHANDLE_UNREGISTERED);
        /*---- perform computation in parallel with data transfer ----*/
        exs_qdequeue(in_qhandle, &in_event, 1, NULL);
        in_bytes = in_event.exs_evt_union.exs_evt_xfer.exs_evt_length;
endloop;
exs_qdelete(out_qhandle);
exs_qdelete(in_qhandle);
exs_close(fd, 0, management_qhandle, NULL);
/*---- perform computation in parallel with EXS activity ----*/
exs_qdequeue(management_qhandle, &management_event, 1, NULL);
exs_qdelete(management_qhandle);
```

10.5. Client using EXS sockets in asynchronous mode with explicit memory registration

Our final step is to combine the changes made independently in the previous two steps, giving us a program that uses both EXS registered memory and EXS asynchronous I/O for RDMA transfers:

```
exs_init(EXS_VERSION);
fd = exs_socket(PF_INET, SOCK_STREAM, 0);
management_qhandle = exs_qcreate(1);
exs_connect(fd, server_address, server_addrlen, 0, NULL, management_qhandle,
                                        NULL);
/*---- perform computation in parallel with EXS activity ----*/
exs_qdequeue(management_qhandle, &management_event, 1, NULL);
in_mhandle = exs_mregister(in_buffer, max_in_bytes, flags);
out_mhandle = exs_mregister(out_buffer, max_out_bytes, flags);
in_qhandle = exs_qcreate(3);
out_qhandle = exs_qcreate(3);
loop
        exs_send(fd, out_buffer, out_bytes, 0, out_qhandle, NULL, out_mhandle);
        /*---- perform computation in parallel with data transfer ----*/
        exs_qdequeue(out_qhandle, &out_event, 1, NULL);
        exs_recv(fd, in_buffer, max_in_bytes, 0, in_qhandle, NULL, in_mhandle);
        /*---- perform computation in parallel with data transfer ----*/
        exs_qdequeue(in_qhandle, &in_event, 1, NULL);
        in_bytes = in_event.exs_evt_union.exs_evt_xfer.exs_evt_length;
endloop;
exs_qdelete(out_qhandle);
exs_qdelete(in_qhandle);
exs_mderegister(out_mhandle);
exs_mderegister(in_mhandle);
exs_close(fd, 0, management_qhandle, NULL);
/*---- perform computation in parallel with EXS activity ----*/
exs_qdequeue(management_qhandle, &management_event, 1, NULL);
exs_qdelete(management_qhandle);
```

This program runs, but as it stands there isn't much parallel activity between the asynchronous EXS activity and the user thread, which should perform parallel computation in the places now marked in the code only by appropriate comments. To take advantage of the potential parallelism, this program needs to be modified to perform useful computation between the call of an **exs_close**() or an **exs_connect**() or an **exs_send**() or an **exs_recv**() that starts an EXS operation and the corresponding **exs_qdequeue**() that waits for the completion of the EXS operation.

11. Status of UNH EXS 1.3.0

11.1. Comparison with the Extended Sockets API (ES-API) Issue 1.0 Specification

UNH EXS function	origin	UNH EXS status	section discussed
exs_accept()	ES-API standard	implemented	4.3.2
exs_bind()	non-standard	implemented	4.3.1
exs_blocking_accept()	non-standard	implemented	9.4
exs_blocking_close()	non-standard	implemented	9.5
exs_blocking_connect()	non-standard	implemented	9.3
exs_blocking_recv()	non-standard	implemented	9.1
exs_blocking_send()	non-standard	implemented	9.1
exs_cancel()	ES-API standard	not implemented	
exs_close()	non-standard	implemented	4.4
exs_connect()	ES-API standard	implemented	4.2
exs_init()	ES-API standard	implemented	3.1
exs_fcntl()	non-standard	implemented	7
exs_listen()	non-standard	implemented	4.3.1
exs_mderegister()	ES-API standard	implemented	8.1
exs_mmodify()	ES-API standard	not implemented	
exs_mregister()	ES-API standard	implemented	8.1
exs_poll()	ES-API standard	not implemented	
exs_qcreate()	ES-API standard	implemented	2.1.1
exs_qdelete()	ES-API standard	implemented	2.1.2
exs_qdequeue()	ES-API standard	implemented	2.1.3
exs_qmodifiy()	ES-API standard	not implemented	
exs_qstatus()	ES-API standard	not implemented	
exs_read()	non-standard	implemented	9.2
exs_recv()	ES-API standard	implemented	5.2
exs_recvmsg()	ES-API standard	not implemented	
exs_send()	ES-API standard	implemented	5.1
exs_sendfile()	ES-API standard	not implemented	
exs_sendmsg()	ES-API standard	not implemented	
exs_socket()	non-standard	implemented	4.1
exs_write()	non-standard	implemented	9.1

11.2. Modifications to the ES-API standard in the UNH EXS implementation

11.2.1. exs.h header file

The definitions of all symbols, structures, and function prototypes introduced by UNH EXS are found in the header file “exs.h”, not “sys/exs.h” as stated in the ES-API standard. Therefore, each “.c” file using UNH EXS should have the following line after all other “include” directives at the beginning of the compilation unit:

#include <exs.h>

11.2.2. exs_accept()

The **EXS_BLOCK** flag has been added in UNH-EXS to indicate that the user wants this function to both start an operation and wait for its completion. When this flag is present, the value of the **qhandle** parameter required in the **exs_accept**() function call is ignored by the EXS interface and can be NULL. The result returned by a successful **exs_accept**() with the **EXS_BLOCK** flag is the **fd** of the new connection to a remote client.

11.2.3. exs_close()

The **EXS_BLOCK** flag has been added in UNH-EXS to indicate that the user wants this function to both start an operation and wait for its completion. When this flag is present, the values of the **qhandle** and **ahandle** parameters that are required in the **exs_close**() function call are ignored by the EXS interface and can be NULL.

11.2.4. exs_connect()

A non-NULL value for the **timeout** parameter, allowed in the ES-API standard, is not yet supported in UNH-EXS.

The **EXS_BLOCK** flag has been added in UNH-EXS to indicate that the user wants this function to both start an operation and wait for its completion. When this flag is present, the values of the **qhandle** and **ahandle** parameters that are required in the **exs_connect**() function call are ignored by the EXS interface and can be NULL.

11.2.5. exs_recv()

The ES-API standard MSG_PEEK and MSG_OOB flag values are not supported in UNH-EXS.

The **EXS_DONTWAIT** flag has been added in UNH-EXS to indicate that the user does NOT want this **exs_recv**() function call to block if there are no available credits for it to start immediately (see section 6.1 above), in which situation the function will immediately return a value of -1 with the value **EAGAIN** stored in the global **errno**.

The **EXS_BLOCK** flag has been added in UNH-EXS to indicate that the user wants this function to both start an operation and wait for its completion. When this flag is present, the values of the **qhandle** and **ahandle** parameters that are required in the **exs_recv**() function call are ignored by the EXS interface and can be NULL.

The **EXS_UNSIGNALED** flag has been added in UNH-EXS to indicate that the user does not want the completion of an asynchronous **exs_recv**() to generate an event. It is ignored when

the **EXS_BLOCK** flag is present. When **EXS_UNSIGNALED** is present, the value of the **qhandle** parameter that is required in the **exs_recv**() function call may be NULL, in which case no event is generated upon the completion of the **exs_recv**(). However, if the value of the **qhandle** parameter is not NULL, an event will be generated in the event queue ONLY if the operation did NOT complete successfully — no event is generated if the operation completed successfully.

11.2.6. exs_send()

The ES-API standard MSG_EOR and MSG_OOB flags are not supported in UNH-EXS.

The **EXS_DONTWAIT** flag has been added in UNH-EXS to indicate that the user does NOT want this **exs_send**() function call to block if there are no available credits for it to start immediately (see section 6.1 above), in which situation the function will immediately return a value of -1 with the value **EAGAIN** stored in the global **errno**.

The **EXS_BLOCK** flag has been added in UNH-EXS to indicate that the user wants this function to both start an operation and wait for its completion. When this flag is present, the values of the **qhandle** and **ahandle** parameters that are required in the **exs_send**() function call are ignored by the EXS interface and can be NULL.

The **EXS_UNSIGNALED** flag has been added in UNH-EXS to indicate that the user does not want the completion of an asynchronous **exs_send**() to generate an event. It is ignored when the **EXS_BLOCK** flag is present. When **EXS_UNSIGNALED** is present, the value of the **qhandle** parameter that is required in the **exs_send**() function call may be NULL, in which case no event is generated upon the completion of the **exs_send**(). However, if the value of the **qhandle** parameter is not NULL, an event will be generated in the event queue ONLY if the operation did NOT complete successfully — no event is generated if the operation completed successfully.

11.3. Known deficiencies

11.3.1. thread cancellation

At the present time, the UNH EXS library functions are NOT cancellation safe, because there are NO cancellation cleanup handlers implemented for any of them. Users are therefore advised NOT to call pthread_cancel() for any of their threads when they might be executing in code using the UNH EXS library.