Winter 2016

Creating a Conformance Testing Framework for the UNH Extended Sockets Library and Demonstrating its Usefulness by Implementing New sendfile() Extension

Maxwell Christopher Renke
University of New Hampshire, Durham

CREATING A CONFORMANCE TESTING FRAMEWORK FOR THE UNH EXTENDED SOCKETS LIBRARY AND DEMONSTRATING ITS USEFULNESS BY IMPLEMENTING NEW SENDFILE() EXTENSION

BY

Maxwell Christopher Renke

B.S., University of New Hampshire, 2015

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

December, 2016

This thesis has been examined and approved in partial fulfillment of the requirements of for the degree of Master of Science in Computer science by:

Thesis Director, Robert Russell, Associate Professor of Computer Science

Phil Hatcher, Associate Professor of Computer Science

Robert Noseworthy, UNH-IOL, Chief Engineer

On December 2, 2016

Original approval signatures are on file with the University of New Hampshire Graduate School.

ACKNOWLEDGMENTS

First and foremost I would like to thank Dr. Robert Russell for allowing me to participate in his research from all the way back to my senior year of my undergraduate degree. I would also like to thank him for his continued technical support and mentorship.

I would like to thank the University of New Hampshire InterOperability Laboratory for giving the opportunity to pursue my academic goals and thrive professionally as well.

I would like to thank Patrick MacArthur for his invaluable technical assistance throughout my research with Dr. Russell and his continued patience with me when I had many, many questions.

Finally, I would like to thank the other members of my thesis committee, Robert Noseworthy and Phil Hatcher, for their continued support.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

CREATING A CONFORMANCE TESTING FRAMEWORK FOR THE UNH EXTENDED SOCKETS LIBRARY AND DEMONSTRATING ITS USEFULNESS BY IMPLEMENTING NEW SENDFILE() EXTENSION

by

Maxwell Christopher Renke

University of New Hampshire, December, 2016

The UNH Extended Sockets Library (UNH EXS) was developed at the University of New Hampshire Interoperability Laboratory to provide an interface to extend the features of the Extended Sockets API (ES-API) specification published by the Open Group to better utilize the asynchronous I/O and memory registration features of Remote Direct Memory Access (RDMA) and provide the programmer with the option to perform operations synchronously as well as asynchronously.

This thesis is focused on building a rigorous testing framework to verify conformance to the published ES-API standards, existing manual pages, and documented UNH extensions of the Extended Sockets Library, and to facilitate regression testing of the software library as a whole. Furthermore, the additional functionality of synchronous and asynchronous sendfile transfer over RDMA with UNH EXS will be implemented, verified, evaluated, and integrated into the existing documentation and testing framework.

The goal of this new capability is to establish a clear process by which new features to UNH EXS can be verified in the future and changes to the library will be properly vetted. The new sendfile transfer functionality is focused on improving the usability and effectiveness of the UNH EXS Library for programmers.

CHAPTER 1

Introduction

1.1 Introduction

The UNH Extended Sockets Library (UNH EXS) [1] provides additional functionality to the Extended Sockets API (ES-API) [2] specification published by the Open Group. The ES-API specification defines extensions to the traditional socket API to provide the asynchronous I/O and memory registration benefits of the Remote Direct Memory Access (RDMA) interface.

UNH EXS implements a large portion of the ES-API standard but also implements unique features to provide the flexibility to choose between both synchronous and asynchronous behavior when using RDMA. UNH EXS also allows the programmer to operate with or without memory registration, a normally mandatory element of using RDMA. Remote Direct Memory Access (RDMA), as the name suggests, allows remote access to user space memory in other RDMA endpoints. RDMA utilizes dedicated hardware interfaces that replace the traditional TCP/IP socket architecture in normal socket programming. This hardware handles both asynchronous I/O operations as well as memory registration and management. During data transfer RDMA bypasses the system kernel, entirely eliminating the need for costly buffer-copying that greatly impacts performance. Thus leveraging the RDMA interface can lead to significant improvement in data throughput.

Programming with RDMA can pose unique challenges compared to traditional socket programming most developers have become accustom to. UNH EXS was created to allow the programmer to use RDMA in a synchronous manner without major overhead while still exposing the asynchronous functionality.

Previously the developers and maintainers of UNH EXS did not have the means to properly

verify the functionality of the software library against the specifications in the ES-API, where applicable, and the documentation created specifically for the UNH EXS library. There is therefore a need for an extensive testing framework to be created that provides both conformance testing of the UNH EXS functions and also regression testing for the maintainers of the UNH EXS library as modifications and additions are made to the library. Furthermore, once a testing framework was developed it needed to be properly vetted. Likewise, documentation created to define the testing framework, both from the end user and developer perspective, also needed to be vetted to ensure the framework and subsequent documentation meet their goals.

Therefore a new feature was added to the UNH EXS library to provide a mechanism to asynchronously (as well as synchronously) transfer files from one RDMA connected endpoint to another. This addition, named `exs_sendfile()` and `exs_blocking_sendfile()`, was designed, specified, and then added to the testing framework to provide a robust way to verify the proper functionality of these functions. The processes described in the documentation accompanying this thesis was followed to ensure future additions to the UNH EXS library would be implemented successfully without disrupting the correct functionality of already implemented features.

CHAPTER 2

Background

2.1 RMDA

Remote Direct Memory Access, or RDMA [3], is a transport that uses asynchronous I/O and memory registration to move data from user space to user space using dedicated hardware and thus bypassing the system kernel entirely. RDMA is widely used in multi-core systems and supercomputers due to the high throughput and low latency that can be achieved.

The two main performance benefits RDMA provides are the lack of context switching between user space and kernel space, as well as eliminating the need to copy buffers between user space and kernel space.

The three main implementations of RDMA hardware interfaces in use today are InfiniBand, iWARP, and RoCE (RDMA over Converged Ethernet). This thesis was written and tested against an InfiniBand implementation. Programs that wish to use RDMA communicate to the hardware device using what is called RDMA Verbs, which are essentially the low level hardware API. These RDMA Verbs are hardware independent.

In order to achieve true performance benefits with RDMA there is significant overhead to ensure the asynchronous operations and memory registration are set up correctly. Achieving this manually requires extensive knowledge of both the hardware platform being developed against as well as a full understanding of how asynchronous operations are handled. Fortunately additional APIs and software libraries have been created to assist in this task.

2.2 OFED Verbs

RDMA Verbs interact directly with the hardware implementations of RDMA. The OpenFabrics Enterprise Distribution (OFED) [7] is an open-source project that defines a set of verbs common to the InfiniBand, iWARP, and RoCE implementations. These verbs can be used to interface directly with the hardware or can be used by libraries such as the ES-API and UNH EXS to perform RDMA operations.

2.3 ES-API

The Extended Sockets API (ES-API) [2] is a specification published by The Open Group with the goal to "provide extensions to the traditional socket API to support improved efficiency in network programming." This means creating an API that uses RDMA to perform asynchronous operations on top of traditional synchronous sockets. The ES-API focuses on the InfiniBand technology and uses traditional POSIX sockets in place of using the RDMA hardware directly. The ES-API operates strictly in an asynchronous manner and an event queue must be managed to handle the completion of different events.

2.4 POSIX

POSIX [4] is a family of standards, specified by the IEEE, that provides a common set of API calls on the system. POSIX style socket programming is starkly different to programming with RDMA. However it has been observed that most programmers familiar with socket programming are familiar with POSIX. There is a significant difference between how one writes a program using POSIX sockets and how one writes a program using RDMA. Most notable is that POSIX sockets rely entirely on the kernel and thus must utilize buffer copying and context switching as well as completely synchronous operations.

2.5 UNH EXS

The University of New Hampshire Interoperability Laboratory has developed the UNH Extended Sockets Library (UNH EXS) to help bridge the gap for programmers familiar with POSIX sockets who wish to use the RDMA protocol effectively. A programmer using UNH EXS can choose whether or not to perform RDMA operations asynchronously or synchronously as well as with or without memory management.

The main difference between UNH EXS and the ES-API is that UNH EXS runs entirely in user space. While the ES-API can rely on the traditional kernel-based socket API, this is not the case for UNH EXS. This is because UNH EXS wishes to be platform independent and thus must not rely on the kernel. Therefore UNH EXS must implement functions that do not appear in the ES-API specification.

```
#include <sys/socket.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]){
        int fd = socket(PF_INET, SOCK_STREAM, 0);
        //use socket here
        close(fd);
}
```

Figure 2-1: Open and Close Socket (ES-API and POSIX)

Figure 2-1 shows how to open and close a socket file descriptor with the ES-API. Figure 2-2 shows how to do the same task but with UNH EXS. The ES-API library relies on the kernel based API calls `socket(2)` [5] and `close(2)` [6] while the UNH EXS example relies entirely on the software library.

UNH EXS supports synchronous operations on top of the existing asynchronous operations of the ES-API. UNH EXS functions that take `int flags` as a parameter can passed `EXS_BLOCK`,

```
#include <exs.h>

int main(int argc, char *argv[]){
        exs_init(EXS_VERSION);
        int fd = exs_socket(PF_INET, SOCK_STREAM, 0);
        //use socket here
        exs_close(fd,0,NULL,NULL);
}
```

Figure 2-2: Open and Close Socket (UNH EXS)

a flag specific to UNH EXS, to specify that the function should operate synchronously (i.e. in blocking mode). If `EXS_BLOCK` is not specified, the function operates in asynchronous mode (i.e. non-blocking mode) as specified in the ES-API.

To demonstrate this, below are two calls to `exs_connect` - one that operates asynchronously and one that operates synchronously.

```
exs_init(EXS_VERSION);
int fd = exs_socket(PF_INET, SOCK_STREAM, 0);
exs_qhandle_t qhandle = exs_qcreate(0);
int ret = exs_connect(fd, info->ai_addr, info->ai_addrlen,
                        0, NULL, qhandle, NULL);
```

Figure 2-3: Asynchronous `exs_connect()`

```
exs_init(EXS_VERSION);
int fd = exs_socket(PF_INET, SOCK_STREAM, 0);
int ret = exs_connect(fd, info->ai_addr, info->ai_addrlen,
                        EXS_BLOCK, NULL, NULL, NULL);
```

Figure 2-4: Synchronous `exs_connect()`

`exs_connect()` in Figure 2-3 operates asynchronously, thus will return immediately and will later post an event to an event queue created by `exs_qcreate()` when the connection has been

created and is ready for use. `exs_qdequeue()` polls from that queue and will wait until it finds the event posted by `exs_connect`. The call to `exs_connect()` in Figure 2-4 will not return until the connection has been established and no event is needed.

It is up to the programmer to properly manage the event queue when using asynchronous operations in UNH EXS just as it is necessary to do so when using the ES-API. The advantage of managing the event queue is that programmers are then able to exploit parallelism in their program design. UNH EXS provides the flexibility to operate without managing the event queue whatsoever.

CHAPTER 3

Related Work

3.1 UNH EXS Overview

The current release of UNH EXS is UNH EXS 1.3.6 [8], published March 15, 2015. This release implements most ES-API operations as well as some non-standard operations. While it is currently possible to establish RDMA connections in each permutation allowed by the library (asynchronous or synchronous, with or without memory registration) there are some features of the ES-API standard that are not implemented.

Documentation detailing each UNH EXS function did not exist when UNH EXS 1.3.6 was published. Quality documentation is required not only for the users of the library to understand how to use the library but also for the developers and maintainers of the library to have strong references when implementing new features or making changes.

At the time of release of UNH EXS 1.3.6 modifying the library required manual testing to ensure the remainder of the library continued to operate as expected. Some trivial test cases were implemented into the build system but were not utilized effectively. The need for a testing framework that could be performed automatically when changes are made to the library was found.

`exs_sendfile()` is a function defined in the ES-API that transmits the contents of a file over a given socket. This functionality is desirable for a user of UNH EXS because sending the contents of a file over an RDMA connection is a common task that requires some overhead to do manually.

The documentation need described above was completed before this thesis (and will be discussed below). However the testing framework and `exs_sendfile()` operation are both the topic of this thesis.

3.2 Manual Pages

Documentation on the syntax and expected behavior of a UNH EXS library did not exist at the time of the UNH EXS 1.3.6 release. A very effective way to provide this documentation is in manual pages implemented directly on the system. Manual pages exist for virtually every UNIX function available to a system and can be accessed directly from the command line using the command "man". Manual pages for each implemented UNH EXS function were created and added to the upcoming release of UNH EXS. These manual pages are now installed on the system when UNH EXS is installed and operate the same way normal manual pages do. Figure 3-1 shows an example of the `exs_connect()` man page, accessed with the command `man exs_connect`.

```
man exs_connect
...
NAME
exs_connect - asynchronously connect a socket

SYNOPSIS
#include   <exs.h>

int        exs_connect(   int                     sockno,
const struct sockaddr   *address,
socklent_t              address_len,
int                     flags,
struct timeval          *timeout,
exs_qhandle_t           qhandle,
exs_ahandle_t           ahandle);

int   exs_blocking_connect(   int                     sockno,
const struct sockaddr   *address,
socklen_t               address_len);

DESCRIPTION
The exs_connect() function initiates a synchronous or asynchronous
connect operation on a socket.

If  the exs_connect() started successfully, it will operate in parallel
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 completion event can be retrieved from the event queue with the
exs_qdequeue() function. The completion event for the exs_connect()
operation is the exs_event structure or equivalent exs_event_t type.

...
```

Figure 3-1: man exs_connect

CHAPTER 4

Testing Framework

4.1 Motivation

The motivation behind creating a testing framework for UNH EXS was twofold. The first was to provide explicit conformance testing for each of the functions defined in UNH EXS whether that be to the ES-API, POSIX, or UNH EXS definition. The second was to provide regression testing when making modifications to the UNH EXS library. Integrating the testing framework into the UNH EXS release allows any user to verify their implementation at any time.

4.2 Conformance Testing

All UNH EXS [1] functions are documented via manual pages (described above) as well as in additional documentation describing UNH EXS. These documents were modeled after the POSIX and ES-API standard when necessary. Thus all conformance testing is based directly off of UNH EXS documentation.

The goal of the conformance testing is to verify that a function produces the correct error codes and behavior in the failure cases and produces the correct return value and behavior in the success cases. Failure cases are mostly testing the various bad parameters that could be passed to a function. The success case is running the function with correct parameters and using successful calls to the functions to assist in the testing of other functions since many UNH EXS functions have pre-requisites.

The framework is built in such a way that if a pre-requisite function fails the other functions that rely on that functions will fail as well. This ensures that if there is an issue with a base level

function or the installation has issues the framework can terminate without doing unnecessary work and not generating a lot of redundant error messages.

4.3 Regression Testing

Regression testing is critical to ensure a software library can mature without introducing bugs to previously functional code. Without regression testing each new change has to be vetted manually and in most cases is not done completely. Success cases are more important than failure cases because failure cases due to parameter checking should only fail if a change is made directly to one function while success cases could fail due to a subtle change in an unrelated function. Having a framework in place to step through all of the possibilities and perform a full test scenario of all functions will ensure fewer errors are introduced into the library as a whole.

Figure 4-1 shows an excerpt from the output of a regression test. `test/test_all_check.sh` is a script that runs all of the test functions and all of their tests. `tests/test_all_check.sh` will return 0 if all tests pass, and 1 if any tests fail. The output of this script can be found in the file `./test-suite.log`.

```
...
PASS: tests/test_all_check.sh
PASS: libexs/test_trace
...
============================================================================
Testsuite summary for UNH EXS 1.3.6-107-gf7d6ff0
============================================================================
# TOTAL: 2
# PASS:  2
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
============================================================================
```

Figure 4-1: Regression Test Output

4.4 Framework Design

Each UNH EXS function has its own set of test cases. These test cases are implemented in a single C program that will parse input parameters, handle initialization, and run the selected test cases. Test cases can be selected from the command line. If no tests are selected, all tests are run. Figure 4-2 shows a simplified example of the test file for `exs_connect()`.

```
#include "result.h"
#include "setup/exs_init_setup.h"
#include "setup/exs_socket_setup.h"
#include "setup/exs_connect_setup.h"
#include "setup/exs_server_setup.h"
#include "setup/exs_client_setup.h"

#define BAD_FLAG 2

void *test_connect(void *test_void_ptr){
        setup_fault_catcher(p,getpid());

        param *p = (param *)test_void_ptr;
        int test = p->test_number;

        /* EPERM: exs_init has not been called */
        if( test == 1 ){
                p->test_value = "EPERM";
                p->test_name = all_connect_tests[test-1];
                int ret = exs_connect(p->fd, p->address,
                p->address_len, p->flags, p->to, p->qh, p->ah);
                print_macro(test,p->test_value,p->test_name,ret,-1,EPERM);
        }

        /** test number not valid **/
        else if( test != 0 ){
                fprintf(stdout,"Invalid Test Number: %d\n", test);
        }
}
int main(int argc, char *argv[]){
        execute(argc,argv,test_connect,CONNECT_TESTS);
}
```

Figure 4-2: Test File for `exs_connect()`

Each test runs in blocking mode by default and can optionally be run in non-blocking mode.

UNH EXS function calls that are limited to the blocking mode (i.e. `exs_blocking_connect()` are not run if non-blocking mode is specified.

A custom `struct param` was created to store information for the various tests. Test Functions take a `struct param` instance as a parameter so that parameters can be passed from test function to test function. `struct param` has an `int mode` field that keeps track of whether or not the testing framework is operating in blocking or non-blocking mode.

`setup_options` handles various command line parameters that specify whether or not passing results are printed, which address and port to use, and which test cases to run. These options are defined in the Testing Framework User Documentation in Appendix A.

```
./tests/test_exs_connect -v 2>/dev/null
1  : EPERM                  : exs_connect(fd, ai_addr, ai_addrlen, flags, to, qh, ah ) [no exs_init]
: blocking     : PASS : 1   : Operation not permitted
2  : EBADF                  : exs_connect(BAD_FD, ai_addr, ai_addrlen, flags, to, qh, ah)
: blocking     : PASS : 9   : Bad file descriptor
3  : EINVAL                 : exs_connect(CONNECTED_FD, ai_addr, ai_addrlen, flags, to, qh, ah)
: blocking     : PASS : 22  : Invalid argument
4  : EINVAL                 : exs_connect(fd, ai_addr, ai_addrlen, BAD_FLAG, to, qh, ah)
: blocking     : PASS : 22  : Invalid argument
5  : OK_EXS_CONNECT         : exs_connect(fd, ai_addr, ai_addrlen, flags, to, qh, ah)
: blocking     : PASS : 0   : Success
./tests/test_exs_connect -v -n 2>/dev/null
1  : EPERM                  : exs_connect(fd, ai_addr, ai_addrlen, flags, to, qh, ah ) [no exs_init]
: non-blocking : PASS : 1   : Operation not permitted
2  : EBADF                  : exs_connect(BAD_FD, ai_addr, ai_addrlen, flags, to, qh, ah)
: non-blocking : PASS : 9   : Bad file descriptor
3  : EINVAL                 : exs_connect(CONNECTED_FD, ai_addr, ai_addrlen, flags, to, qh, ah)
: non-blocking : PASS : 22  : Invalid argument
4  : EINVAL                 : exs_connect(fd, ai_addr, ai_addrlen, BAD_FLAG, to, qh, ah)
: non-blocking : PASS : 22  : Invalid argument
5  : OK_EXS_CONNECT         : exs_connect(fd, ai_addr, ai_addrlen, flags, to, qh, ah)
: non-blocking : PASS : 0   : Success
./tests/test_exs_blocking_connect -v 2>/dev/null
1  : EPERM                  : exs_blocking_connect(fd, ai_addr, ai_addrlen) [no exs_init]
: blocking     : PASS : 1   : Operation not permitted
2  : EINVAL                 : exs_blocking_connect(CONNECTED_FD, ai_addr, ai_addrlen)
: blocking     : PASS : 22  : Invalid argument
3  : OK_EXS_BLOCKING_CONNECT : exs_blocking_connect(fd, ai_addr, ai_addrlen)
: blocking     : PASS : 0   : Success
```

Figure 4-3: Sample Output for `test_exs_connect`

`print_macro` is used in the testing framework to determine the result of a test based on the expected return value versus the observed return value as well as the observed error code. If the return value and error code match what the test has specified, the result is a pass. Otherwise, the result is a fail. All results are printed to Standard Out. Passing results are not printed unless the command line options specify verbose output. Figure 4-3 shows sample output for

`test_exs_connect` which tests `exs_connect()`. Note that `2>/dev/null` is used to eliminate prints to `stderr` as they are not relevant in this example.

The testing framework was designed to operate on a single machine and create a connection on a single local interface. This is done to ensure there are no synchronization issues between two machines trying to establish a connection. To do this, the testing framework uses two `FIFOs`, one to open a connection and one to close it, as a mutex to ensure synchronous (blocking) calls to UNH EXS functions avoid a race condition. These `FIFOs` are not used in asynchronous (non-blocking) mode.

The tests in a test file mostly check the parameter validation of each function. After parameter checks, the test file tests correct calls to the EXS function. This is done through helper functions with the prefix `ok`. For example, Figure 4-4 shows `ok_exs_connect()`.

```
static int ok_exs_connect(param *p)
{
        char *name = "ok_exs_connect";

        struct addrinfo *info;
        struct addrinfo hints;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = p->family;
        hints.ai_socktype = p->socktype;
        hints.ai_protocol = p->protocol;
        hints.ai_flags = AI_PASSIVE;

        int r = getaddrinfo(p->addr, p->port, &hints, &info);

        int ret =
        exs_connect(p->fd, info->ai_addr, info->ai_addrlen, p->flags,
        p->to, p->qh, p->ah);

        if (ret < 0) {
                print_error(name);
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

Figure 4-4: `ok_exs_connect()` helper function

Note that `ok_exs_connect()` does not initialize UNH EXS (via `exs_init()`) nor does it close the connection it establishes with `exs_connect()`. The `ok` functions are helper functions to allow test cases to be set up more easily.

These `ok` functions are combined to set up the test cases into the various states required to test a specific function. A full connection between a client and a server can be initialized, connected, and closed using only these functions (as seen in Figure 4-5). If any of these helper functions fails, the test case immediately terminates and fails as well.

4.5 Framework Documentation

The operation of the UNH EXS testing framework is documented in a separate User Documentation document that is included in this thesis as Appendix A. The User Documentation also details each UNH EXS function and its test cases.

The implementation of the UNH EXS testing framework is documented in a separate Developer Documentation document that is included in this thesis as Appendix B. The Developer Documentation is focused on three things: exposing the specification and expected behavior of the UNH EXS functions, providing the user with the knowledge of how to run the testing framework and interpret the results, and providing developers who maintain UNH EXS and the testing framework a detailed explanation on how the testing framework is implemented. This is done to ensure that new test cases and test functions can be added easily as the library is extended.

4.6 Integration into EXS Build System

The testing framework has been integrated into the UNH EXS build system. Once the appropriate version of UNH EXS is installed the testing framework can built and run on the system.

Each UNH EXS function test is compiled into a single script that will be run by the UNH EXS build process. The results from each test are displayed to Standard Output (by default tests that pass are suppressed) and these results can be checked by the build process to determine if there

were any failure cases. At this time if a single test cases fails the build process will state a failure of all tests as a whole but the user can view the output and determine which specific test case failed. The test cases can also be run independently of the UNH EXS build process once they are built.

```
#include "result.h"

int main(int argc, char *argv[]){
        p = calloc(1,sizeof(param));
        p->func = "pingpong";

        /** get command line options **/
        if (setup_options(argc, argv, p) != 0) {
                return 0;
        }

        pid_t pid = fork();
        if( pid == 0 ){
                if( ok_exs_init(p) < 0 ) exit(EXIT_FAILURE);
                if( ok_exs_socket(p) < 0 ) exit(EXIT_FAILURE);
                if( ok_exs_bind(p) < 0 ) exit(EXIT_FAILURE);
                if( ok_exs_listen(p) < 0 ) exit(EXIT_FAILURE);

                int fd = open("fifo1",O_WRONLY);

                if( ok_exs_accept(p) < 0 ) exit(EXIT_FAILURE);

                int ret = ok_exs_recv(p);
                if( ret < p->RECV_BUFSIZE ) exit(EXIT_FAILURE);

                close(fd);

                if( ok_exs_close(p) < 0 ) exit(EXIT_FAILURE);

                fd = open("fifo2", O_RDONLY);

                close(fd);
        } else {
                int fd = open("fifo1",O_RDONLY);

                if( ok_exs_init(p) < 0 ) exit(EXIT_FAILURE);
                if( ok_exs_socket(p) < 0 ) exit(EXIT_FAILURE);
                if( ok_exs_connect(p) < 0 ) exit(EXIT_FAILURE);

                close(fd);

                if( ok_exs_send(p) < 0 ) exit(EXIT_FAILURE);
                if( ok_exs_close(p) < 0 ) exit(EXIT_FAILURE);

                fd = open("fifo2", O_WRONLY);

                close(fd);
        }
}
```

Figure 4-5: Client and Server Connection

CHAPTER 5

Asynchronous File Transfer

5.1 Motivation

Transmitting the contents of a file across a socket is a common task. While it is possible to do this with UNH EXS 1.3.6 using normal send operations these commands can only send buffers of memory in user space. In order to transmit a file the programmer would have to go through the overhead of bringing the contents of file into memory and then use send operations to transmit the data. Figure 5-1 demonstrates this task where the user simply reads in parts of a file block by block and transmits these blocks with `exs_send()`.

Bringing the file contents into memory one block at a time can be inefficient. Figure 5-2 demonstrates how to avoid sending the file in separate blocks and instead map the entire contents of the file into memory using `mmap(2)` [9]. This is the way `exs_sendfile()` is implemented.

The ES-API defines `exs_sendfile()` as a function to transmit the contents of a file across a socket. UNH EXS does not implement this function. Making this addition to the UNH EXS library serves two purposes. First it adds additional functionality that is desired and brings UNH EXS closer to a complete implementation of the ES-API `exs_sendfile()`. Secondly it allows the testing framework to be vetted as a useful means of performing test-first development and also provide regression testing as a new feature is added. The documentation described in Appendix B will also be used and evaluated on how it assists in making additions to the testing framework.

```
#include <exs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

/* main takes a single argument, the path of a
file to send using exs_sendfile */
        int main(int argc, char* argv[]){
        if( argc < 2 ) return 0;

        exs_init(EXS_VERSION);

        int sfd = exs_socket(PF_INET, SOCK_STREAM, 0);

        int family = PF_INET;
        int socktype = SOCK_STREAM;
        int protocol = IPPROTO_TCP;
        struct addrinfo *info;
        struct addrinfo hints;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = family;
        hints.ai_socktype = socktype;
        hints.ai_protocol = protocol;
        hints.ai_flags = AI_PASSIVE;

        int r = getaddrinfo("localhost", "55555", &hints, &info);
        exs_connect(sfd, info->ai_addr, info->ai_addrlen,
        EXS_BLOCK, NULL, NULL, NULL);

        ssize_t block = (ssize_t)1024;
        ssize_t size;
        char buf[block];

        int fd = open(argv[1],O_RDONLY);

        exs_mhandle_t mh = exs_mregister(buf,size,
        EXS_MRF_RECV_DISABLE);

        while ((size = read(filefd, &buf, block)) > 0) {
                exs_send(p->fd, buf, size, EXS_BLOCK, NULL, NULL, mh);
        }

        exs_mderegister(mh, 0);

        exs_close(sfd, EXS_BLOCK, NULL, NULL);
        close(fd);
}
```

Figure 5-1: Sending a file with `exs_send()` in blocks

```
#include <exs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

/* main takes a single argument, the path of a
file to send using exs_sendfile */
int main(int argc, char* argv[]){
        if( argc < 2 ) return 0;

        exs_init(EXS_VERSION);

        int sfd = exs_socket(PF_INET, SOCK_STREAM, 0);

        int family = PF_INET;
        int socktype = SOCK_STREAM;
        int protocol = IPPROTO_TCP;
        struct addrinfo *info;
        struct addrinfo hints;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = family;
        hints.ai_socktype = socktype;
        hints.ai_protocol = protocol;
        hints.ai_flags = AI_PASSIVE;

        int r = getaddrinfo("localhost", "55555", &hints, &info);
        exs_connect(sfd, info->ai_addr, info->ai_addrlen,
        EXS_BLOCK, NULL, NULL, NULL);

        ssize_t size;
        struct stat stat_buf;

        int fd = open(argv[1],O_RDONLY);

        fstat(fd, &stat_buf);
        size = stat_buf.st_size;

        void* mapped = mmap(0,size,PROT_READ | PROT_WRITE,
        MAP_PRIVATE,fd,offset);

        exs_mhandle_t mh = exs_mregister(mapped,(size_t)size,
        EXS_MRF_RECV_DISABLE);
        exs_send(sfd, mapped, size, EXS_BLOCK, NULL, NULL, mh);
        exs_mderegister(mh, 0);

        munmap(mapped, size);

        exs_close(sfd, EXS_BLOCK, NULL, NULL);
        close(fd);
}
```


Figure 5-2: Sending a file with `exs_send()` with `mmap(2)`

5.2 Benefits

`exs_sendfile()` and its companion `exs_blocking_sendfile()` reduces a common task in RDMA programming into a well defined function. This reduces overhead and complexity on the part of the programmer wishing to complete this task. Once this mechanism is in place the UNH EXS framework will provide a more rich API to the user as well as achieve closer parity with the ES-API `exs_sendfile()`.

5.3 Description

`exs_sendfile()` was not implemented exactly as it appears in the ES-API standard. First, it supports synchronous operation, i.e. the function will not return until the entire file is transmitted over the socket or an error occurs. Second, due to time constraints, `exs_sendfile()` is limited to accepting files that are less than or equal to 1GB in length. This was done because 1GB is the maximum allowed message size supported by the InfiniBand hardware UNH EXS was implemented on. Third, a new mechanism for specifying which file to transmit was added.

As specified in the ES-API, `exs_sendfile()` in UNH EXS only accepts a valid file descriptor that points to regular file (i.e. not a socket, fifo, etc.). See Figure 5-3 for a complete example.

```
#include <exs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>

/* main takes a single argument, the path of a
file to send using exs_sendfile */
int main(int argc, char* argv[]){
        if( argc < 2 ) return 0;

        exs_init(EXS_VERSION);
        int sfd = exs_socket(PF_INET, SOCK_STREAM, 0);

        int family = PF_INET;
        int socktype = SOCK_STREAM;
        int protocol = IPPROTO_TCP;

        struct addrinfo *info;
        struct addrinfo hints;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = family;
        hints.ai_socktype = socktype;
        hints.ai_protocol = protocol;
        hints.ai_flags = AI_PASSIVE;

        int r = getaddrinfo("localhost", "55555", &hints, &info);
        exs_connect(sfd, info->ai_addr, info->ai_addrlen,
        EXS_BLOCK, NULL, NULL, NULL);

        int fd = open(argv[1],O_RDONLY);

        exs_xferfile_t sendvec;
        memset(&sendvec, 0, sizeof(exs_xferfile_t));

        sendvec.exs_xfv.exs_fdv.exs_fildes = fd;
        sendvec.exs_xfv.exs_fdv.exs_offset = (off_t)0;
        sendvec.exs_xfv.exs_fdv.exs_length = (size_t)0;
        sendvec.exs_xfv.exs_fdv.exs_flags = 0;
        sendvec.exs_xfv_type = EXS_FDVEC;

        int sendvec_cnt = 1;

        exs_sendfile(sfd, &sendvec, sendvec_cnt, EXS_BLOCK, NULL, NULL);
        exs_close(sfd, EXS_BLOCK, NULL, NULL);
        close(fd);
}
```

Figure 5-3: `exs_sendfile()` with valid file descriptor

```
union exs_xfvec
{
        exs_iovec_t     exs_iov;                /* IOVEC extent. */
        exs_fdvec_t     exs_fdv;                /* FDVEC extent. */
        exs_pathvec_t   exs_pathv;              /* PATHVEC extent. */
};

struct exs_xferfile
{
        int             exs_xfv_type;   /* Source extent type. */
        union exs_xfvec exs_xfv;        /* Source extent. */
};
```

Figure 5-4: `exs_xferfile` and `exs_xfvec`

```
struct exs_pathvec
{
        off_t           exs_offset;     /* Offset into the file. */
        size_t          exs_length;     /* Requested transfer length. */
        int             exs_flags;      /* Flags. */
        char            *exs_path;      /* path to file */
};
```

Figure 5-5: `exs_pathvec`

UNH EXS will eventually be ported to Windows where file descriptors have a different format than in POSIX. Therefore it was desired to include another means of specifying a file by giving it the name of a file path. The file path will be checked by `exs_sendfile()` and the function will return in error if the path is not valid (or the resulting file does not meet the other requirements). This is achieved by adding a new `struct`, `exs_pathvec`, to the `exs_xfvec union`, which is defined in `struct exs_xferfile` (see Figure 5-4). `exs_xferfile` is what is passed to `exs_sendfile()` and now contains either a file descriptor (`exs_fdvec`) or a file path (`exs_pathvec`). `exs_pathvec` is defined in Figure 5-5 and is used in Figure 5-6.

```
#include <exs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char* argv[]){
        if( argc < 2 ) return 0;

        exs_init(EXS_VERSION);
        int sfd = exs_socket(PF_INET, SOCK_STREAM, 0);

        int family = PF_INET;
        int socktype = SOCK_STREAM;
        int protocol = IPPROTO_TCP;

        struct addrinfo *info;
        struct addrinfo hints;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = family;
        hints.ai_socktype = socktype;
        hints.ai_protocol = protocol;
        hints.ai_flags = AI_PASSIVE;

        int r = getaddrinfo("localhost", "55555", &hints, &info);
        exs_connect(sfd, info->ai_addr, info->ai_addrlen,
        EXS_BLOCK, NULL, NULL, NULL);

        exs_xferfile_t sendvec;
        memset(&sendvec, 0, sizeof(exs_xferfile_t));

        sendvec.exs_xfv.exs_pathv.exs_path = argv[1];
        sendvec.exs_xfv.exs_pathv.exs_offset = (off_t)0;
        sendvec.exs_xfv.exs_pathv.exs_length = (size_t)0; //go until EOF
        sendvec.exs_xfv.exs_pathv.exs_flags = 0; //EXS_SHUT_WR
        sendvec.exs_xfv_type = EXS_PATHVEC;

        int sendvec_cnt = 1;

        exs_sendfile(sfd, &sendvec, sendvec_cnt, EXS_BLOCK, NULL, NULL);
        exs_close(sfd, EXS_BLOCK, NULL, NULL);
}
```

Figure 5-6: `exs_sendfile()` with `exs_pathvec`

```
EXS_EXPORT ssize_t
exs_blocking_sendfile(int sockno, exs_xferfile_t *sendvec, int sendvec_cnt,
int flags)
{
        return exs_sendfile(sockno, sendvec, sendvec_cnt, flags|EXS_BLOCK,
                                NULL, NULL);
}       /* exs_blocking_sendfile */
```

Figure 5-7: `exs_blocking_sendfile()`

As for other synchronous functions, `exs_blocking_sendfile()` (see Figure 5-7) will be added to the UNH EXS library as an alias to `exs_sendfile()` in blocking mode, but with the irrelevant parameters removed.

5.4 Challenges

`exs_sendfile()` must operate in synchronous and asynchronous mode as well as optionally support memory registration. Additionally there is overhead in verifying the file descriptor or file path passed to the function. Fortunately many of these challenges are addressed by `exs_send()`, the normal send operation in UNH EXS, and `exs_send()` can be used as a model to implement `exs_sendfile()`.

The other challenge with creating `exs_sendfile()` is if the file size is greater than the supported message size defined by the RDMA hardware UNH EXS is implemented on. This would require `exs_sendfile()` to bring the file into memory in pieces and transmit those pieces independently. This would mean keeping track of each iteration (as a failure of one iteration would mean a failure of the entire file transmission) as well as the operation as a whole. This challenge is avoided by creating the requirement that `exs_sendfile()` must not support a file over 1GB so that it can be sent in its entirety as one message using a single `exs_send()` call.

5.5 Design

`exs_sendfile()` is modeled after `exs_send()`. In fact, they will share the same core code - sending a buffer of memory over a socket. `exs_sendfile()` has additional overhead to check the various parameters and to verify the file specified meets the necessary requirements.

In the case where `exs_sendfile()` is given a valid file descriptor, `mmap(2)` [9] is used to map the file data into memory and `exs_mregister()` is used to register the memory for sending. Once the send has been completed, `exs_mderegister()` and `munmap(2)` [9] are used to unregister and free the file data in memory.

In the case where `exs_sendfile()` is given a file path rather than a file descriptor the function will attempt to open the file and return an error if that operation fails. If it succeeds it will attempt to send the file over the socket following the same steps as in the file descriptor case. When the operation is complete `exs_sendfile()` will close the file specified.

5.6 Specification

`exs_sendfile()` and `exs_blocking_sendfile()` are defined in the User Documentation (Appendix A) on page 30.

CHAPTER 6

Conclusion

6.1 Lessons Learned

The testing framework became an invaluable tool while implementing `exs_sendfile()` and `exs_blocking_sendfile()`. Not only was the various various parameter checking able to be verified easily, but it was easy to add additional test cases to ensure the behavior of the functions were correct. Additionally, running the test framework as a whole during the implementation process provided regression testing that ensured that `exs_sendfile` would not break something else in the library.

Creating documentation, both for the user and developers of the testing framework, was also very useful. It helped detect issues in the design as well as the implementations of the various test functions and test cases. It also helped to ensure that the testing framework could be extended in the future.

Additions to a software library are never easy and this thesis was a good opportunity to make a substantial contribution to the library and also allow for easier extension in the future.

6.2 Summary

The contents of this thesis are included in the UNH EXS 1.4.0 [10]. This includes the creation of a testing framework and the implementation of two functions: `exs_sendfile()` and `exs_blocking_sendfile()`. These functions were added to the testing framework which was used to vet their implementation. User and Developer documentation for the testing framework were created and are attached to this thesis.

6.3 Future Work

A natural expansion to `exs_sendfile()` (and `exs_blocking_sendfile()`) would be to allow file sizes over 1GB. Additionally, the functions `exs_recvfile()` and `exs_blocking_recvfile` could be added to receive data from a socket and place it into a file. `exs_sendfile()` and `exs_recvfile()` combined would allow for true end to end file transfer.

APPENDIX A

User Documentation for UNH EXS 1.4.0 Testing Framework

Maxwell Renke
InterOperability Laboratory
University of New Hampshire
Durham, New Hampshire 03824

1. Introduction

This document describes an non-exhaustive suite of tests that will run nightly against the development installation of UNH EXS 1.4.0 to provide a detailed report on the current status of the UNH EXS development and support regression testing for future development.

The tests described in this document will include unit tests against the **Extended Sockets API** (ES-API) specification published by the **OpenGroup** as well as existing **man page** documentation included with the 1.4.0 UNH EXS installation.

The end result of this test suite will issues reports detailing failing tests. This mechanism is further described in **Framework**.

2. Scope

This document will describe the layout, method, and reporting techniques used to support regression testing for the UNH EXS 1.4.0 installation. The main goal of this test suite is to directly verify the behavior of UNH EXS functions to the the UNH EXS documentation as well as ES-API conformance.

3. Framework

Test cases will be structured in such a way that they contain prerequisite tests that must pass in order to be run. For example, if **Initialization** fails then the failure will be reported followed by an error message stating that all further tests will not be run because of the failure. Test cases may be performed and reported on in a different order than they appear on this document.

Test cases will be individually reported on in each group. Test groups will maintain a running tally of successful test cases as well as failure test cases. Any failures will be displayed in-line with the rest of the testing, unless such a failure requires the entire test group to not be tested.

An example test report is included at the end of this document.

A test result shall include **test group number, test number, function name, return value, errno, result, message** that can be independently verified.

The following contains an overview, in order, on how the tests are structured such that each test succeeds a test that is required to pass before it can be tested.

3.1 Initialization

exs_init(), exs_socket()

3.2 Client Connection

exs_bind()

3.2.1 Asynchronous

exs_connect()

3.2.2 Synchronous

exs_blocking_connect()

3.3 Listening Post

exs_listen()

3.3.1 Asynchronous

exs_accept()

3.3.2 Synchronous

exs_blocking_accept()

3.4 Memory Registration

exs_mderegister(), exs_mregister()

3.5 Completion Queues

exs_qcreate(), exs_qdelete(), exs_qdequeue()

3.6 Message Transfer

3.6.1 Asynchronous

exs_send(), exs_receive(), exs_sendfile()

3.6.2 Synchronous

exs_blocking_send(), exs_blocking_recv(), exs_read(), exs_write(), exs_blocking_sendfile()

3.7 Socket Termination

exs_shutdown()

3.7.1 Asynchronous

exs_close()

3.7.2 Synchronous

exs_blocking_close()

3.8 Other

exs_fcntl()

4. Command Line Usage

The UNH EXS Testing Framework runs as a command line utility. The results can be used to support Regression Testing, which will be discussed in detail in the following subsections. The options available to the user as well as detailed information for the test cases that can be run can be viewed directly from the command line. Shell scripts are used to run the tests in aggregate.

4.1 Command Line Options

The following options can be viewed from the command line by using the **[-h]** or **[-H]** option on the command line at run time.

Command line options and their processing are defined in **cmdline.h**.

The options that will be displayed will be as follows:

[–h –H]	Displays usage information.
[–l –L]	Displays all tests for a particular function, including the test number, test name, and expected return value.
[–t –T]	Displays an individual test for a particular function, including the test number, test name, and expected return value when **followed by a number**.
[–v –V]	Specifies the option for Verbose output – all passing and non passing tests will be printed to **stdout**. By default, passing results are not printed.
[–a –A]	Specifies the server interface address to be used in the testing framework. By default, this value is set to “localhost”.
[–p –P]	Specifies the port number to be used in the testing framework. By default, this value is hard-coded in **port.h**.
[–n –N]	Specifies the option to run the testing framework in **non-blocking (asynchronous) mode**. By default, the testing framework runs in **blocking mode**.
[0]	Specify which test to run. Can be separated by spaces. Invalid or numbers out of range of the test will not be accepted. If not test is selected, all tests will be run.
[0-1]	Specify a range of tests to run. Invalid ranges will not be accepted.

4.1.1 Individual Tests

Each EXS function can be run independently of each other. To run a particular test it is necessary to compile the project and run the appropriate object file with the appropriate command line options. For example:

```
./test_exs_init –v
```

This will run all **exs_init** tests in **verbose** output mode.

4.1.2 Running All Tests

The UNH EXS Framework includes the bash script file **test_all.sh**. This simple runs all test functions with the parameters to the script passed to each function.

For example:

```
./test_all.sh –v –p 55555
```

This will run all tests in **verbose** mode with a **port number** of 55555. Note, the project must be compiled before **test_all.sh** can be run.

4.2 Regression Testing

The UNH EXS Framework reports to **stdout** as well as a **log** file. Each line includes the test number, test name, expected result, observed result, and test result. It shall be sufficient to **diff** different outputs from running the framework to determine if the results have changed and if so in what way. The mechanism to achieve true automated regression testing is not in the scope of this document and is left as an exercise for the reader.

5. Test Groups

The tests described in this document are separated into various groups to better segment the initialization required for each tests and to keep results consistent across groups.

The groups include:

- ES-API Conformance
- EXS Conformance

5.2. ES-API Conformance

The goals of these tests are to verify that all implemented ES-API functions correctly conform to specification published by the **OpenGroup**.

These tests will attempt to:

- Verify all functions return **0** on success cases.
- Verify all functions return **-1** on failure cases.
- Verify all functions set **errno** appropriately on failure cases.

Undefined behavior will be reported as failures.

5.2.1. exs_accept()

The following tests will exercise exs_accept() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind(), exs_listen(), exs_connect(), exs_qcreate(), exs_qdequeue()

```
int exs_accept(      int                    socket,
                     exs_acceptaddr_t       *addrvec,
                     int                    addrvec_cnt,
                     int                    flags,
                     exs_qhandle_t          qhandle);
```

`socket`	file descriptor used to specify the socket to be used
`addrvec`	pointer to array of `exs_acceptaddr` structures
`addrvec_cnt`	specifies the number of connections that should be accepted
`flags`	additional options to be specified (blocking/non-blocking)
`qhandle`	specifies the destination event queue

Test [1]: EPERM

```
exs_accept(p->fd, &acceptaddr, 1, EXS_BLOCK, NULL) [no exs_init]
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_accept(p->fd, &acceptaddr, 1, BAD_FLAG, NULL)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flag** is invalid

Test [3]: EINVAL

```
exs_accept(p->fd, &acceptaddr, NEGATIVE_ADDRVEC_CNT, 0, qh)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **addrvec_cnt** <= 0 when non blocking
- In **blocking** mode, this test case is not performed.

Test [4]: EINVAL

```
exs_accept(p->fd, &acceptaddr, ZERO_ADDRVEC_CNT, EXS_BLOCK, qh)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **addrvec_cnt** != 1 when blocking
- In **non-blocking** mode, this test case is not performed.

Test [5]: EINVAL

```
exs_accept(p->fd, &acceptaddr, 1, 0, INVALID_QHANDLE)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **qhandle** is not a valid event queue.

Test [6]: EINVAL

```
exs_accept(p->fd, &acceptaddr, 0, EXS_BLOCK, NULL) [no exs_listen]
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if the socket pointed to by **fd** is not accepting connections (**exs_listen()** has not been called)

Test [6]: EBADF

```
exs_accept(BAD_FD, &acceptaddr, 1, EXS_BLOCK, NULL)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **fd** is invalid

Test [8]: OK_EXS_ACCEPT

```
exs_accept(p->fd, &acceptaddr, 1, EXS_BLOCK, NULL)
```

- Verify the function returns **0** if passed correct parameters.

5.2.2. exs_connect()

The following tests will exercise exs_connect() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind(), exs_listen(), exs_qcreate(), exs_qdequeue()

```
int exs_connect(     int                     sockno,
                     const struct sockaddr   *address,
                     socklent_t              address_len,
                     int                     flags,
                     struct timeval          *timeout,
                     exs_qhandle_t           qhandle,
                     exs_ahandle_t           ahandle);
```

`sockno`	file descriptor used to specify the socket to be used
`address`	points to a `sockaddr` structure containing the peer address
`address_len`	specifies the length of the `sockaddr` structure pointed to by `address`
`flags`	additional options to be specified (blocking/non-blocking)
`timeout`	specifies how long the asynchronous connect operation should take before timing out
`qhandle`	specifies the destination event queue
`ahandle`	arbitrary pointer value chose by the user

Test [1]: EPERM

```
exs_connect(fd, ai_addr, ai_addrlen, flags, to, qh, ah) [no exs_init]
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_connect(fd, ai_addr, ai_addrlen, -1, to, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flags** is invalid (i.e. set to -1).

Test [3]: EINVAL

```
exs_connect(fd, ai_addr, ai_addrlen, BAD_FLAG, to, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flags** is invalid.

Test [4]: EBADF

```
exs_connect(BAD_FD, ai_addr, ai_addrlen, flags, to, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]** if the **socket** argument is not a valid file descriptor.

Test [5]: EINVAL

```
exs_connect(CONNECTED_FD, ai_addr, ai_addrlen, flags, to, qh, ah)
```

- Verify the function returns **-1** and sets **errno**to **[EINVAL]** if the **socket** provided is already connected or listened to.

Test [6]: OK_EXS_CONNECT

```
exs_connect(fd, ai_addr, ai_addrlen, flags, to, qh, ah)
```

- Verify the function returns **0** if passed correct parameters.

5.2.3. exs_init()

The following tests will exercise exs_init() and its various success and error cases.

Prerequisites:

```
int exs_init(        int                        version);
```

`version` requested version of the API to be initialized

Test [1]: ENOTSUP

```
exs_init(INVALID_VERSION)
```

- Verify the function returns **-1** and sets **errno** to **[ENOTSUP]** if the **version** parameter provided to **exs_init()** is not supported.

Test [2]: OK_EXS_INIT

```
exs_init(VERSION)
```

- Verify the function returns **0** if passed a correct **version** parameter.

Test [3]: EALREADY

```
exs_init(VERSION) [exs_init already]
```

- Verify the function returns **-1** and sets **errno** to **[EALREADY]** if **exs_init()** has already been called in the same process.

5.2.4. exs_mderegister()

The following tests will exercise exs_mderegister() and its various success and error cases.

Prerequisites: exs_init()

```
exs_mhandle_t   exs_mderegister( exs_mhandle_t                mhandle,
                                 int                          flags);
```

`mhandle`	specifies the handle to the memory being deregistered
`flags`	additional options to be specified

Test [1]: EPERM

```
exs_mderegister(mh, 0)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_mderegister(NULL, 0)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **mhandle** is **NULL**.

Test [3]: EINVAL

```
exs_mderegister(EXS_MHANDLE_INVALID, 0)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **mhandle** is set to **EXS_MHANDLE_INVALID**.

Test [4]: EINVAL

```
exs_mderegister(EXS_MHANDLE_UNREGISTERED, 0)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **mhandle** is set to **EXS_MHANDLE_UNREGISTERED**.

Test [5]: OK_EXS_MDEREGISTER

```
exs_mderegister(mh, 0)
```

- Verify the function returns **0** if passed correct parameters.

5.2.5. exs_mregister()

The following tests will exercise exs_mregister() and its various success and error cases.

Prerequisites: exs_init()

```
exs_mhandle_t   exs_mregister(   void                          *addr,
                                 size_t                        size,
                                 int                           flags);
```

`addr`	address to the application memory to be registered
`size`	length of the application memory to be registered
`flags`	additional options to be specified

Test [1]: EPERM

```
exs_mregister(buf, BUFSIZE, 0)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3]: EINVAL

```
exs_mregister(buf, ZERO_SIZE, 0)
```

- Verify the function returns **EXS_MHANDLE_INVALID** and sets **errno**to**[EINVAL]** if **size** is less than or equal to **0**.

Test [4]: EINVAL

```
exs_mregister(buf, BUFSIZE, BAD_FLAG)
```

- Verify the function returns **EXS_MHANDLE_INVALID** and sets **errno**to**[EINVAL]** if **flag** is invalid.

Test [5]: EINVAL

```
exs_mregister(NULL_ADDR, BUFSIZE, 0)
```

- Verify the function returns **EXS_MHANDLE_INVALID** and sets **errno**to**[EINVAL]** if **addr** is **NULL**.

Test [6]: OK_EXS_MREGISTER

```
exs_mregister(buf, BUFSIZE, 0)
```

- Verify the function returns a **valid opaque memory handle** if passed correct parameters.

Unimplemented Tests:

- Verify the function returns **EXS_MHANDLE_INVALID** and sets **errno**to**[EACCES]** if the function fails due to **memory permissions**.

5.2.6. exs_qcreate()

The following tests will exercise exs_qcreate() and its various success and error cases.

Prerequisites: exs_init()

```
exs_qhandle_t   exs_qcreate(     int                depth);
```

`depth` specifies the guaranteed minimum number of events that can be stored in the queue.

Test [1]: EPERM

```
exs_qcreate(10)
```

- Verify the function returns **EXS_QHANDLE_INVALID** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3]: OK_EXS_QCREATE

```
exs_qcreate(10)
```

- Verify the function returns a **valid event queue** if passed correct parameters.

Unimplemented Tests:

- Verify the function returns **EXS_QHANDLE_INVALID** and sets **errno** to**[EAGAIN]** if the allocation of internal resources **failed** but a subsequent request may succeed.
- Verify the function returns **EXS_QHANDLE_INVALID** and sets **errno** to**[EINVAL]** if the event queue resources were **exceeded**.

5.2.7. exs_qdelete()

The following tests will exercise exs_qdelete() and its various success and error cases.

Prerequisites: exs_init(), exs_qcreate()

```
int exs_qdelete(     exs_qhandle_t                  qhandle);
```

`qhandle` specifies the event queue to be deleted

Test [1]: EPERM

```
exs_qdelete(qh)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3]: EINVAL

```
exs_qdelete(INVALID_QHANDLE)
```

- Verify the function returns **-1** and sets **errno**to **[EINVAL]** if **qhandle** is not a valid event queue.

Test [4]: OK_EXS_QDELETE

```
exs_qdelete(qh)
```

- Verify the function returns **0** if passed a valid **qhandle**.

Unimplemented Tests:

- Verify the function returns **-1** and sets **errno** to **[EBUSY]** if there are still asynchronous operations in progress for this event queue.

5.2.8. exs_qdequeue()

The following tests will exercise exs_qdequeue() and its various success and error cases.

Prerequisites: exs_init(), exs_qcreate()

```
int exs_qdequeue(          exs_qhandle_t                qhandle,
                           exs_event_t                  *evtvec,
                           int                          evtvec_cent,
                           const struct timeval         *timeout);
```

`qhandle`	specifies an event queue
`*evtvec`	specifies the address of an array of `exs_event_t` types
`evtvec_cent`	specifies the number of `exs_event_t` elements in the `evtvec` array
`timeout`	specifies how long the call should wait before timing out

Test [1]: EPERM

```
exs_qdequeue(qh, evtvec, evtvec_cent,to)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_qdequeue(INVALID_QH, evtvec, evtvec_cent,to)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3]: EINVAL

```
exs_qdequeue(qh, evtvec, ZERO_EVTVEC_CENT,to)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **evtvec_cnt** is less than 1.

Unimplemented Tests:

- Verify the function returnsthe correct number of **dequeued events** if passed correct parameters.
- Verify the function returns **-1** and sets **errno** to **[EFAULT]** if:
  - The address range specified by **evtvec** and **evtvec_cnt** is not valid.
  - **timeout** is invalid
- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **evtvec_cnt** is greater than **EXS_EVTVEC_MAX**.

5.2.9. exs_recv()

The following tests will exercise exs_recv() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind(), exs_listen(), exs_accept()

```
ssize_T exs_recv(          int                 socket,
                           void                *buffer,
                           size_t              length,
                           int                 flags
                           exs_qhandle_t       qhandle,
                           exs_ahandle_t       ahandle,
                           exs_mhandle_t       mhandle);
```

`socket`	specifies the socket file descriptor
`buffer`	points to a buffer where the message should be stored
`length`	specifies the length in bytes of the buffer pointed to by the `buffer` argument
`flags`	specifies additional options (blocking/non-blocking)
`qhandle`	specifies an event queue
`ahandle`	specifies an arbitrary pointer value chosen by the user
`mhandle`	specifies a registered memory handle

Test [1]: EPERM

```
exs_recv(fd, recv_buf, RECV_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_recv(fd, recv_buf, RECV_BUFSIZE, flags, INVALID_QH, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **qhandle** is invalid.

Test [3]: EINVAL

```
exs_recv(fd, recv_buf, RECV_BUFSIZE, 0, INVALID_QH, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **qhandle** is **NULL** and **EXS_BLOCK** is not specified.

Test [4]: EINVAL

```
exs_recv(fd, recv_buf, ZERO_LENGTH, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **length** is not at least 1.

Test [5]: EINVAL

```
exs_recv(fd, recv_buf, MAX_LENGTH, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **length** is greater than the maximum supported length for the RDMA channel.

Test [6]: EINVAL

```
exs_recv(fd, BAD_BUFFER, RECV_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **buffer** is **NULL**.

Test [7]: EOPNOTSUPP

```
exs_recv(fd, recv_buf, RECV_BUFSIZE, BAD_FLAG, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EOPNOTSUPP]** if at least one of the **flags** passed to the function are not supported.

Test [8]: EBADF

```
exs_recv(BAD_FD, recv_buf, RECV_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EBADF]** if the function was passed a bad file descriptor.

Test [9]: ENOTCONN

```
exs_recv(BAD_FD, recv_buf, RECV_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[ENOTCONN]** if the socket was never connected to prior to the function call.

Test [10]: OK_EXS_RECV

```
exs_recv(BAD_FD, recv_buf, RECV_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **0** if passed all valid parameters.

5.2.10. exs_send()

The following tests will exercise exs_send() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_connect()

```
ssize_t exs_send(          int                     socketno,
                           const void              *buffer,
                           size_t                  length,
                           int                     flags,
                           exs_qhandle_t           qhandle,
                           exs_ahandle_t           ahandle,
                           exs_mhandle_t           mhandle);
```

`socket`	specifies the socket file descriptor
`buffer`	points to a buffer containing the message to send
`length`	specifies the length in bytes of the buffer pointed to by the `buffer` argument
`flags`	specifies additional options (blocking/non-blocking)
`qhandle`	specifies an event queue
`ahandle`	specifies an arbitrary pointer value chosen by the user
`mhandle`	specifies a registered memory handle

Test [1]: ENOTCONN

```
exs_send(NOT_CONNECTED_FD, send_buf, SEND_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[ENOTCONN]** if the socket was never connected.

Test [2]: EPERM

```
exs_send(fd, send_buf, SEND_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3]: EINVAL

```
exs_send(fd, send_buf, SEND_BUFSIZE, flags, INVALID_QHANDLE, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **qhandle** is invalid.

Test [4]: EINVAL

```
exs_send(fd, send_buf, SEND_BUFSIZE, 0, INVALID_QHANDLE, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **qhandle** is **NULL** and **EXS_BLOCK** is not specified and **qhandle** is **NULL** and E**XS_UNSIGNALED** is not specified.

Test [5]: EINVAL

```
exs_send(BAD_BUFFER, send_buf, SEND_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **buffer** is **NULL**.

Test [7]: EOPNOTSUPP

```
exs_send(fd, send_buf, SEND_BUFSIZE, BAD_FLAG, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EOPNOTSUPP]** if **flags** is not supported.

Test [8]: EBADF

```
exs_send(BAD_FD, send_buf, SEND_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EBADF]** if the function was passed a bad file descriptor.

Test [10]: EPIPE

```
exs_send(fd, send_buf, SEND_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EPIPE]** if the connection has been gracefully shutdown for writing.

Test [16]: OK_EXS_SEND

```
exs_send(fd, send_buf, SEND_BUFSIZE, flags, qh, ah, mh)
```

- Verify the function returns **0** if passed all valid parameters.

Unimplemented Tests:

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[ECONNRESET]** if the connection has been terminated abruptly.
- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **EXS_UNSIGNALED** and **EXS_BLOCK** were both specified.
- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EBUSY]** if no send credits were available at the time of this function call

5.2.11. exs_sendfile()

The following tests will exercise exs_sendfile() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_connect()

```
ssize_t exs_sendfile(      int                    socket,
                           exs_xferfile_t         *sendvec,
                           int                    sendvec_cnt,
                           int                    flags,
                           exs_qhandle_t          qhandle,
                           exs_qhandle_t          ahandle);
```

`socket`	specifies the socket file descriptor
`sendvec`	specifies an array of file descriptors and memory buffers
`sendvec_cnt`	specifies the number of elements in `sendvec` array
`flags`	specifies additional options (blocking/non-blocking)
`qhandle`	specifies an event queue
`ahandle`	specifies an arbitrary pointer value chosen by the user

Test [1]: EPERM

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: ENOTCONN

```
exs_sendfile(NOT_CONNECTED_FD, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[ENOTCONN]** if the socket was never connected.

Test [3]: EINVAL

```
exs_sendfile(BAD_FD, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]** if **fd** is invalid.

Test [4]: ENODEV

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[ENODEV]** if the **file descriptor** cannot be mmapped.

Test [5]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **exs_fileds** is invalid.

Test [6]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **exs_path** is invalid.

Test [7]: EFBIG

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EFBIG]** if **file size > 1GB**.

Test [8]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **offset + length > fize size**.

Test [9]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **offset > file size**.

Test [10]: EINVAL

```
exs_sendfile(fd, sendvec, BAD_SENDVEC_CNT, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **sendvec_cnt** is invalid.

Test [11]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, BAD_FLAG, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flags** is invalid.

Test [12]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, EXS_UNSIGNALED & EXS_BLOCK, qh,
ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **EXS_UNSIGNALED** and **EXS_BLOCK** were both specified.

Test [13]: EINVAL

```
exs_sendfile(fd, sendvec, sendvec_cnt, EXS_BLOCK, INVALID_QHANDLE, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **EXS_BLOCK** is specified and **qhandle** is invalid.

Test [14]: EPIPE

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **-1** and sets **errno** to **[EPIPE]** if the socket is gracefully shut down for writing.

Test [15]: OK_EXS_SENDFILE

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **0** if passed all valid parameters.

Test [16]: OK_EXS_SENDFILE

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **0** if passed all valid parameters and exs_sendfile() is used to send the same file immediately afterward.

Test [15]: OK_EXS_SENDFILE

```
exs_sendfile(fd, sendvec, sendvec_cnt, flags, qh, ah);
```

- Verify the function returns **0** if passed all valid parameters using **exs_pathvec** rather than **exs_fdvec**.

5.3. EXS Conformance

The goals of these tests are to verify that all EXS functions that are not part of the ES-API specification correctly conform to documentation provided by the UNH EXS 1.4.0 installation.

These tests will attempt to:

- Verify all functions return **0** on success cases.
- Verify all functions return **-1** on failure cases.
- Verify all functions set **errno** appropriately on failure cases.

Undefined behavior will be reported as failures.

5.3.1 exs_bind()

The following tests will exercise exs_bind() and its various success and error cases.

Prerequisites: exs_init(), exs_socket()

```
int exs_bind(        int                     fd,
                     struct sockaddr         *address,
                     socketlen_t             addrlen);
```

`fd`	specifies socket file descriptor
`address`	pointer to a structure of type `struct sockaddr` that contains port address and number information
`addrlen`	size in bytes of the structure pointed to by `address`

Test [1]: EPERM

```
exs_bind(fd, ai_addr, ai_addrlen)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EBADF

```
exs_bind(BAD_FD, ai_addr, ai_addrlen)
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]** if **fd** is invalid.

Test [3]: EINVAL

```
exs_bind(CONNECTED_FD, ai_addr, ai_addrlen)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if the socket provided by **fd** has already been bound.

Test [4]: OK_EXS_BIND

```
exs_bind(fd, ai_addr, ai_addrlen)
```

- Verify the function returns **0** if passed correct parameters.

5.3.2 exs_blocking_accept()

The following tests will exercise exs_blocking_accept() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind(), exs_listen()

```
int exs_blocking_accept(   int                    sockno,
                           struct sockaddr        *addr,
                           socklen_t              *addrlen);
```

`sockno`	specifies socket file descriptor
`addr`	points to an array of `exs_acceptaddr` structures
`addrlen`	specifies the number of connections that should be accepted

Test [1]: EPERM

```
exs_blocking_accept(fd, addr_store, addrlen)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_blocking_accept(fd, addr_store, addrlen)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **socket** is not accepting connections (**exs_listen()** has not been called).

Test [3]: OK_EXS_BLOCKING_ACCEPT

```
exs_blocking_accept(fd, addr_store, addrlen)
```

- Verify the function returns the **socket** for the new connection to the remote client if passed correct parameters.

Unimplemented Tests:

- Verify the function returns **-1** and sets **errno**to **[EINVAL]** if:
  - **flags** is not supported
  - **addrvec_cnet** is less than or equal to 0
  - **qhandle** is not a valid event queue
- Verify the function returns **-1** and sets **errno** to **[EBADF]** if the **socket** argument is not a valid file descriptor.

5.3.3 exs_blocking_close()

The following tests will exercise exs_blocking_close() and its various success and error cases.

Prerequisites: exs_init(), exs_socket()

```
int exs_blocking_close(    int                         sockno);
```

`sockno` specifies socket file descriptor

Test [1]: EPERM

```
exs_blocking_close(fd)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: OK_EXS_BLOCKING_CLOSE

```
exs_blocking_close(fd)
```

- Verify the function returns **0** and correctly closes the socket if passed the correct parameters.

Unimplemented Tests:

- Verify the function returns **-1** and sets **errno** to **[EINVAL]**if:
  - **flags** are not supported
  - **qhandle** is invalid
- Verify the function returns **-1** and sets **errno** to **[EBADF]** if **socket** is invalid.

5.3.4 exs_blocking_connect()

The following tests will exercise exs_blocking_connect() and its various success and error cases.

Prerequisites: exs_init(), exs_socket()

```
int exs_blocking_connect(  int                          sockno,
                           const struct sockaddr        *address,
                           socklen_t                    address_len);
```

`sockno`	specifies socket file descriptor
`address`	points to a `sockaddr` structure containing the peer address
`address_len`	specifies the length of the `sockaddr` structure pointed to by the `address` argument

Test [1]: EPERM

```
exs_blocking_connect(fd, ai_addr, ai_addrlen, flags, mh, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_blocking_connect(CONNECTED_FD, ai_addr, ai_addrlen, flags, mh, qh,
ah)
```

- Verify the function returns **-1** and sets **errno**to **[EINVAL]** if the **socket** provided is already connected or listened to.

Test [3]: OK_EXS_BLOCKING_CONNECT

```
exs_blocking_connect(CONNECTED_FD, ai_addr, ai_addrlen, flags, mh, qh,
ah)
```

- Verify the function returns **0** if passed correct parameters.

5.3.5 exs_blocking_recv()

The following tests will exercise exs_blocking_recv() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind(), exs_listen(), exs_accept()

```
ssize_t exs_blocking_recv( int              sockno,
                           void             *buffer,
                           size_t           max_bytes,
                           int              flags,
                           exs_mhandle_t    mhandle);
```

`sockno`	specifies socket file descriptor
`buffer`	points to a buffer where the message should be stored
`max_bytes`	specifies length in bytes of the buffer point to by the `buffer` argument
`flags`	specifies additional options
`mhandle`	specifies a registered memory handle

Test [1]: EPERM

```
exs_blocking_recv(fd, recv_buf, RECV_BUFSIZE, flags, mh)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: EINVAL

```
exs_blocking_recv(fd, recv_buf, ZERO_LENGTH, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EINVAL]** if **length** is not at least 1.

Test [3]: EINVAL

```
exs_blocking_recv(fd, recv_buf, MAX_LENGTH, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EINVAL]** if **length** is greater than the maximum supported length for the RDMA channel.

Test [4]: EINVAL

```
exs_blocking_recv(fd, BAD_BUFFER, RECV_BUFSIZE, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EINVAL]** if **buffer** is **NULL**.

Test [5]: EOPNOTSUPP

```
exs_blocking_recv(fd, recv_buf, RECV_BUFSIZE, BAD_FLAG, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EOPNOTSUPP]** if at least one of the **flags** passed to the function are not supported.

Test [6]: EBADF

```
exs_blocking_recv(BAD_FD, recv_buf, RECV_BUFSIZE, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EBADF]** if the function was passed a bad file descriptor.

Test [7]: ENOTCONN

```
exs_blocking_recv(NOT_CONNECTED_FD, recv_buf, RECV_BUFSIZE, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[ENOTCONN]** if the socket was never connected to prior to the function call.

Test [8]: OK_EXS_BLOCKING_RECV

```
exs_blocking_recv(fd, recv_buf, RECV_BUFSIZE, flags, mh)
```

- Verify the function returns the number of bytes successfully received if passed the correct parameters.

5.3.6 exs_blocking_send()

The following tests will exercise exs_blocking_send() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_connect()

```
ssize_t exs_blocking_send(      int             sockno,
                                void            *buffer,
                                size_t          max_bytes,
                                int             flags,
                                exs_mhandle     mhandle);
```

`sockno`	specifies socket file descriptor
`buffer`	points to a buffer where the message should be stored
`max_bytes`	specifies length in bytes of the buffer point to by the `buffer` argument
`flags`	specifies additional options
`mhandle`	specifies a registered memory handle

Test [1]: ENOTCONN

```
exs_blocking_send(NOT_CONNECTED_FD, send_buf, SEND_BUFSIZE, flags, mh)
```

- Verify the function returns **-1** and sets **errno** to **[ENOTCONN]** if the socket was never connected.

Test [2]: EPERM

```
exs_blocking_send(fd, send_buf, SEND_BUFSIZE, flags, mh)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3]: EINVAL

```
exs_blocking_send(fd, BAD_BUFFER, SEND_BUFSIZE, flags, mh)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **buffer** is **NULL**.

Test [4]: EINVAL

```
exs_blocking_send(fd, send_buf, SEND_BUFSIZE, EXS_UNSIGNALED |
EXS_BLOCK, mh)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **EXS_UNSIGNALED** and **EXS_BLOCK** were both specified.

Test [5]: EOPNOTSUPP

```
exs_blocking_send(fd, send_buf, SEND_BUFSIZE, BAD_FLAG, mh)
```

- Verify the function returns **-1** and sets **errno** to **[EOPNOTSUPP]** if **flags** is not supported.

Test [6]: EBADF

```
exs_blocking_send(BAD_FD, send_buf, SEND_BUFSIZE, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EBADF]** if the function was passed a bad file descriptor.

Test [8]: EBADF

```
exs_blocking_send(fd, send_buf, SEND_BUFSIZE, flags, mh)
```

- Verify the function returns **-1**and sets **errno** to **[EPIPE]** if the connection has been gracefully shutdown for writing.

Test [14] : OK_EXS_BLOCKING_SEND

```
exs_blocking_send(fd, send_buf, SEND_BUFSIZE, flags, mh)
```

- Verify the function returns the number of bytes successfully transmitted if passed the correct parameters.

Unimplemented Tests:

- Verify the function returns **-1** and sets **errno** to **[ECONNRESET]** if the connection has been terminated abruptly.
- Verify the function returns **-1** and sets **errno** to **[EBUSY]** if no send credits were available at the time of this function call.

5.3.7 exs_close()

The following tests will exercise exs_close() and its various success and error cases.

Prerequisites: exs_init(), exs_socket()

```
int exs_close(       int                    sockno,
                     int                    flags,
                     exs_qhandle_t          qhandle,
                     exs_ahandle_t          ahandle);
```

`sockno`	specifies the socket file descriptor
`flags`	specifies additional options (blocking/non-blocking)
`qhandle`	specifies an event queue
`ahandle`	specifies an arbitrary pointer value chosen by the user

Test [1] : EPERM

```
exs_close(fd, flags, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2] : EINVAL

```
exs_close(fd, BAD_FLAG, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flags** are not supported.

Test [3] : EINVAL

```
exs_close(fd, flags, INVALID_QHANDLE, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **qhandle** is invalid.

Test [4] : EINVAL

```
exs_close(BAD_FD, flags, qh, ah)
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]** if **socket** is invalid.

Test[5] : OK_EXS_CLOSE

```
exs_close(fd, flags, qh, ah)
```

- Verify the function returns **0** and an asynchronous event is posted to **qhandle** if passed the correct parameters.

5.3.8 exs_listen()

The following tests will exercise exs_listen() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind()

```
int exs_listen(      int               fd,
                     int               backlog);
```

`fd`	specifies socket file descriptor
`backlog`	specifies maximum number of outstanding client connections

Test [1] : EPERM

```
exs_listen(fd, 10)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2] : EINVAL

```
exs_listen(fd, -1)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]**if **backlog** is negative.

Test [3] : EINVAL

```
exs_listen(CONNECTED_FD, 10)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]**if **fd** specifies a socket that has already been connected or listened to.

Test [4] : EBADF

```
exs_listen(CONNECTED_FD, 10)
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]**if **fd** is invalid.

Test [5] : OK_EXS_LISTEN

```
exs_listen(fd, 10)
```

- Verify the function returns **0** if passed correct parameters.

5.3.9 exs_read()

The following tests will exercise exs_read() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_bind(), exs_listen(), exs_accept()

```
int exs_read(       int                     sockno,
                    void                    *buffer
                    size_t                  max_bytes);
```

`sockno`	specifies socket file descriptor
`buffer`	points to a buffer where the message should be stored
`max_bytes`	specifies length in bytes of the buffer point to by the `buffer` argument

Test [1] : EPERM

```
exs_read(fd, recv_buf, RECV_BUFSIZE)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2] : EINVAL

```
exs_read(fd, recv_buf, ZERO_LENGTH)
```

- Verify the function returns **-1**and sets **errno** to **[EINVAL]** if **length** is not at least 1.

Test [3] : EINVAL

```
exs_read(fd, recv_buf, MAX_LENGTH)
```

- Verify the function returns **-1**and sets **errno** to **[EINVAL]** if **length** is greater than the maximum supported length for the RDMA channel.

Test [4] : EINVAL

```
exs_read(fd, INVALID_QHANDLE, RECV_BUFSIZE)
```

- Verify the function returns **-1**and sets **errno** to **[EINVAL]** if **qhandle** is **NULL**.

Test [5] : EBADF

```
exs_read(BAD_FD, recv_buf, RECV_BUFSIZE)
```

- Verify the function returns **-1**and sets **errno** to **[EBADF]** if the function was passed a bad file descriptor.

Test [6] : ENOTCONN

```
exs_read(NOT_CONNECTED_FD, recv_buf, RECV_BUFSIZE)
```

- Verify the function returns **-1**and sets **errno** to **[ENOTCONN]** if the socket was never connected to prior to the function call.

Test [7] : OK_EXS_READ

```
exs_read(fd, recv_buf, RECV_BUFSIZE)
```

- Verify the function returns the number of bytes successfully received if passed the correct parameters.

5.3.10 exs_shutdown()

The following tests will exercise exs_shutdown() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_connect(), exs_bind(), exs_listen(), exs_accept()

```
int exs_shutdown(          int                     sockno,
                           int                     how,
                           int                     flags,
                           exs_qhandle_t           qhandle,
                           exs_qhandle_t           ahandle);
```

`sockno`	specifies socket file descriptor
`how`	specifies which direction the connection will be shut down
`flags`	additional options to be specified (blocking/non-blocking)
`qhandle`	specifies the destination event queue
`ahandle`	arbitrary pointer value chose by the user

Test [1] : EPERM

```
exs_shutdown(fd, SHUT_RDWR, flags, qh, qh)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2] : EINVAL

```
exs_shutdown(fd, SHUT_RDWR, BAD_FLAG, qh, qh)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flags** is invalid.

Test [3] : EINVAL

```
exs_shutdown(fd, SHUT_RDWR, flags, INVALID_QHANDLE, qh)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **qhandle** is invalid.

Test [4] : EBADF

```
exs_shutdown(BAD_FD, SHUT_RDWR, flags, qh, qh)
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]** if **sockno** is invalid.

Test [5] : OK_EXS_SHUTDOWN

```
exs_shutdown(fd, SHUT_RDWR, flags, qh, qh)
```

- Verify the function returns **0** and an asynchronous event is posted to **qhandle** if passed the correct parameters.

5.3.11 exs_socket()

The following tests will exercise exs_socket() and its various success and error cases.

Prerequisites: exs_init()

```
int exs_socket(      int               family,
                     int               socktype,
                     int               protocol);
```

`family`	must be `PF_INET` or `PFINET6`
`socktype`	must be `SOCK_STREAM` or `SOCK_SEQPACKET`
`protocol`	must be 0 or match the `socktype`

Test [1] : EPERM

```
exs_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [16] : OK_EXS_SOCKET

```
exs_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)
```

- Verify the function returns a non-negative integer file descriptor **fd** if passed correct parameters.

Test [2] – [16] :

- Verify the function returns **-1** and sets **errno** to **[EAFNOSUPPORT]** if **family** is not supported.

5.3.12 exs_write()

The following tests will exercise exs_write() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_connect()

```
ssize_t exs_write(          int                 sockno,
                            void                *buffer,
                            size_t              max_bytes);
```

`sockno`	specifies socket file descriptor
`buffer`	points to a buffer where the message should be stored
`max_bytes`	specifies length in bytes of the buffer point to by the `buffer` argument

Test [1] : ENOTCONN

```
exs_write(NOT_CONNECTED_FD, send_buf, SEND_BUFSIZE)
```

- Verify the function returns **-1** and sets **errno** to **[ENOTCONN]** if the socket was never connected.

Test [2] : EPERM

```
exs_write(fd, send_buf, SEND_BUFSIZE)
```

- Verify the function returns **-1** and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [3] : EINVAL

```
exs_write(fd, BAD_BUFFER, SEND_BUFSIZE)
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **buffer** is **NULL**.

Test [5] : EPIPE

```
exs_write(fd, send_buf, SEND_BUFSIZE)
```

- Verify the function returns **-1**, and sets **errno** to **[EPIPE]** if the connection has been gracefully shutdown for writing.

Test [11] : OK_EXS_WRITE

```
exs_write(fd, send_buf, SEND_BUFSIZE)
```

- Verify the function returns the number of bytes successfully transmitted if passed the correct parameters.

Unimplemented Tests:

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[ECONNRESET]** if the connection has been terminated abruptly.
- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EINVAL]** if **EXS_UNSIGNALED** and **EXS_BLOCK** were both specified.
- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EBUSY]** if no send credits were available at the time of this function call

5.3.13. exs_blocking_sendfile()

The following tests will exercise exs_blocking_sendfile() and its various success and error cases.

Prerequisites: exs_init(), exs_socket(), exs_connect()

```
ssize_t exs_sendfile(      int                    socket,
                           exs_xferfile_t         *sendvec,
                           int                    sendvec_cnt,
                           int                    flags);
```

`socket`	specifies the socket file descriptor
`sendvec`	specifies an array of file descriptors and memory buffers
`sendvec_cnt`	specifies the number of elements in `sendvec` array
`flags`	specifies additional options (blocking/non-blocking)

Test [1]: EPERM

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[EPERM]** if **exs_init()** has not yet been called in the same process.

Test [2]: ENOTCONN

```
exs_blocking_sendfile(NOT_CONNECTED_FD, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1**, no event is posted to **qhandle**, and sets **errno** to **[ENOTCONN]** if the socket was never connected.

Test [3]: EINVAL

```
exs_blocking_sendfile(BAD_FD, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EBADF]** if **fd** is invalid.

Test [4]: ENODEV

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[ENODEV]** if the **file descriptor** cannot be mmapped.

Test [5]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **exs_fileds** is invalid.

Test [6]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **exs_path** is invalid.

Test [7]: EFBIG

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EFBIG]** if **file size > 1GB**.

Test [8]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **offset + length > fize size**.

Test [9]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **offset > file size**.

Test [10]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, BAD_SENDVEC_CNT, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **sendvec_cnt** is invalid.

Test [11]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, BAD_FLAG);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **flags** is invalid.

Test [12]: EINVAL

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, EXS_UNSIGNALED &
EXS_BLOCK);
```

- Verify the function returns **-1** and sets **errno** to **[EINVAL]** if **EXS_UNSIGNALED** and **EXS_BLOCK** were both specified.

Test [13]: EPIPE

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **-1** and sets **errno** to **[EPIPE]** if the socket is gracefully shut down for writing.

Test [14]: OK_EXS_BLOCKING_SENDFILE

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **0** if passed all valid parameters.

Test [16]: OK_EXS_ BLOCKING_SENDFILE

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **0** if passed all valid parameters and exs_blocking_sendfile() is used to send the same file immediately afterward.

Test [15]: OK_EXS_ BLOCKING_SENDFILE

```
exs_blocking_sendfile(fd, sendvec, sendvec_cnt, flags);
```

- Verify the function returns **0** if passed all valid parameters using **exs_pathvec** rather than **exs_fdvec**.

5.4. EXS Integration

The goals of these tests are to verify the correctness of various important combinations of UNH EXS requests and functions. Some of this behavior is defined by the ES-API specification while other behavior described by documentation provided by the UNH EXS 1.4.0 installation.

These tests are currently implemented in a limited number (e.g. non-exhaustive).

6. EXS Functions

The following table lists all UNH EXS functions and provides reference to applicable test cases within this document.

6.1 Applicable Test Cases for ES-API and EXS Conformance

UNH EXS Function	Origin	UNH EXS Status	Applicable Test Cases
exs_accept()	ES-API standard	implemented	5.2.1
exs_bind()	non-standard	implemented	5.3.1
exs_blocking_accept()	non-standard	implemented	5.3.2
exs_blocking_close()	non-standard	implemented	5.3.3
exs_blocking_connect()	non-standard	implemented	5.3.4
exs_blocking_recv()	non-standard	implemented	5.3.5
exs_blocking_send()	non-standard	implemented	5.3.6
exs_blocking_sendfile()	non-standard	implemented	5.3.13
exs_close()	non-standard	implemented	5.3.7
exs_connect()	ES-API standard	implemented	5.2.2
exs_init()	ES-API standard	implemented	5.2.3
exs_listen()	non-standard	implemented	5.3.8
exs_mderegister()	ES-API standard	implemented	5.2.4
exs_mregister()	ES-API standard	implemented	5.2.5
exs_qcreate()	ES-API standard	implemented	5.2.6
exs_qdelete()	ES-API standard	implemented	5.2.7
exs_qdequeue()	ES-API standard	implemented	5.2.8
exs_read()	non-standard	implemented	5.3.9
exs_recv()	ES-API standard	implemented	5.2.9
exs_send()	ES-API standard	implemented	5.2.10
exs_sendfile()	ES-API standard	implemented	5.3.11
exs_shutdown()	non-standard	implemented	5.3.10
exs_socket()	non-standard	implemented	5.3.11
exs_write()	non-standard	implemented	5.3.12

APPENDIX B

Developer Documentation for UNH EXS Testing Framework

Maxwell Renke
InterOperability Laboratory
University of New Hampshire
Durham, New Hampshire 03824

1. Introduction

The goal of this document is to define the layout, structure, and functionality of the UNH EXS Testing Framework and provide a means for UNH EXS developers to maintain and update the framework as well as add additional test cases and functionality to the testing framework as a whole.

2. Structure

The UNH EXS framework is structured into key **include** files, **setup** files, and the individual **test case** files. These will be expanded and enumerated on below. Each test case utilizes the same framework defined by command files and only the individual test cases will change within the **test case** files.

2.1 Include Files

These files contain content that will be necessary for each **test case** file. In particular they cover common functions, parameters, and structs that contain test specific parameters. While all tests for a particular UNH EXS function is defined in the same file each execution of the test is run by the framework itself. The framework handles initialization, execution, reporting, and tear-down of each test. The mechanisms and operations on how the framework operates will be explained in further sections.

2.1.1 param.h

This file holds the **param** structure used to store and pass parameters and other values between the framework and the test functions. This includes normal data types as well as EXS specific structures. This document does not contain an exhaustive list of the kinds of parameters used in the framework, but listed below are some important or otherwise noteworthy parameters essential to the framework.

The **param** structure is initialized before any tests are run and the parameters are set with default values or with values specified by the user.

func	The name of the function being tested. Initialized immediately after **param** is initialized.
mode	Specifies if the framework is running in **blocking (synchronous)** or **non-blocking (asynchronous) mode**.
addr	Specifies the server interface address to be used in the testing framework. It is defaulted to a hard-coded value and can be specified by the user from the command line.
port	Specifies the port number to be used in the testing framework. It is defaulted to a hard-coded value and can be specified by the user from the command line.
test_name	String usually associated with the function being tested as it appears in the code with the relevant parameters displayed to show the user what is being tested.
test_value	String that denotes the expected result of the test. Usually denotes the error code that is expected, or the "OK" result.
test_number	Integer that denotes the test number associated with the test that is used when a specific test is specified by the user. This value is also printed with every test output.
Iter	Linked list node that is used to iterate through the various test cases for a particular function.
flags	UNH EXS flag options values that are passed to the individual test functions during testing. The value of flags will differ based on the operation mode of the framework (blocking vs. non-blocking).

2.1.2 fault.h

This file defines a **fault handler** in order to catch various test-terminating events such as SIGSEV, SIGABRT, and SIGALRM. This allows the test cases to terminate gracefully and report their result in the final output.

Additionally, SIGALRM is used to safeguard against tests that do not halt. SIGALRM will be raised after a fixed amount of time for each test and the result of such a signal will be reported in the final output.

2.1.3 print.h

This file contains various functions to aid in printing results (and other messages) to **stdout** and the **log** file. **print_error** and **print_message** are simple helper methods while **print_macro** and the other **_macro** functions handle printing the full results when a test is terminated.

2.1.3.1 print_error

```
static int print_error(char* name)
```

This method prints the string **"A correct call to %s failed. The test suite will not terminate.\n"** to **stdout**. **%s** refers to **name**.

2.1.3.2 print_mesasge

```
static int print_message(char* msg)
```

This method prints **msg** to **stdout** if the **VERBOSE** flag is set to **1** and prints **msg** to the **log** file regardless.

2.1.3.3 print_macro

```
static int print_macro(int test, char* exp, char* name, int
ret_value, int ret_expected, int ret_errno)
```

This method takes several parameters and prints them to **stdout** if the **VERBOSE** flag is set to **1** and prints **msg** to the **log** file regardless. The mode of operation (**blocking** or **non-blocking**) is also printed. See **Results** in section 3.

2.1.3.4 print_mhandle_macro

```
static int print_mhandle_macro(int test, char* exp, char* name,
exs_mhandle_t ret_value, int ret_errno)
```

This method operates in the same way as **print_macro** except that **ret_value** is of the **exs_mhandle_t** type.

2.1.3.5 print_qhandle_macro

```
static int print_qhandle_macro(int test, char* exp, char* name,
exs_qhandle_t ret_value, int ret_errno)
```

This method operates in the same way as **print_macro** except that **ret_value** is of the **exs_qhandle_t** type.

2.1.4 cmdline.h

This file defines, maintains, and processes the various command line options available to the user. **cmdline.h** also initializes the **param** struct to be used in each test case file. This file also builds the linked list elements needed to specify which test numbers will be run. The details on the various command line options can be found in the UNH EXS User Documentation.

2.1.4.1 setup_options

This method handles the command line options that the user may or may not specify as well as initializing parameters in the **param struct**. In particular, by default, the parameters are initialized to values that will set the framework to run in **blocking mode**. Note that by default, the **address** and **port** values are hardcoded to values – they must be edited for the specific machine the framework is running on, or specified in the command line every time.

Other than the specified command line options, the user can specify tests that need to be run either **individually** or in a **range**. These values are then parsed by **setup_options** and a linked list is created which is then used when the test is executed.

2.1.5 result.h

This file contains most of the common includes, variables, and method definitions that are used by all of the test files. For example, it includes the other include files mentioned above, plus the following methods:

run_test(int i, void* test_func)

This method will run a test specified by **i** which is defined by the **test_func** pointer to be a general way of running an individual test. This method will create a child process to run the test and the **param struct** will be passed to the new thread. This method all sets up the SIGALARM that is used to terminate the process started by **run_test** if it runs too long.

list_tests(param *local_p, int i)

This method is a generalized mapping of a function to print out the tests available for each function to the command line. Each test defines its own method in its **Setup Files**.

execute(int argc, char *argv[], void* test_func, int num_tests)

This method is what is passed to **run_test** and defines the tests that need to be run when the user enters a command. It initializes the **param struct**, calls **setup_options**, then either lists the tests specified, or runs all tests if no test number is specified, or runs the specified tests in the order they were received.

2.2 Setup Files

These files define reference functions to be used throughout the test framework as calls to the functions with valid parameters that should return success if the implementation the framework is run against is functioning properly. These are referred to as **OK** functions. These functions are also used to set variables in the **param struct** that other functions will use. For example, **ok_exs_socket** will set **fd** to a valid socket address.

These files defines an array of test **names** that are used in the **print_macro** call.

These files define the **list_function_tests** that will print out the **test numbers** as well as **test names**.

2.2.1 OK Functions

OK test functions are functions that call a function with valid parameters in both **blocking** and **non-blocking** mode (when applicable). These functions return success and failure, but do not report their own results. **OK** test functions are used as **prerequisites** for other tests. If an **OK** test function fails, a specific **error message** will be reported. In general, if an **OK** function fails the current test should terminate (or will fail due to parameters that were meant to be initialized were not).

2.3 Test Files

Test files must follow the following format:

```
void *test_FUNCTION(void *test_void_ptr){
    setup_fault_catcher(p,getpid());

    param *p = (param *)test_void_ptr;
    int test = p->test_number;


     if( test == 1 ){
          /** test code goes here **/
     }

    /** test number not valid **/
    else if( test != 0 ){
        fprintf(stdout,"Invalid Test Number: %d\n", test);
    }
}

int main(int argc, char *argv[]){
    execute(argc,argv,test_send,FUNCTION_TESTS);
}
```

Note: **FUNCTION** should be replaced with the name of the applicable function (excluding EXS)

Note that **test** is initialized to the **test_number** value in the **param struct** that was initialized before the test was actually called. Each test will be defined in its own block and the details of the code necessary for a test to complete will be expanded upon later in this document.

Each test also includes a call to the **OK** function call (defined in the **setup** file related to the test files), usually at the end of the test file. This test will be a call with valid parameters in both **blocking** and **non-blocking** mode (when applicable).

2.4 Test Blocks

This section will attempt to describe the necessary elements to a **test block** in the **testing framework**. It will describe **key functions** that must be included, as well as how to utilize the **param struct**, **blocking** versus **non-blocking** mode, and using the **print_macro** properly.

2.4.1 Initialization

Two **param struct** variables must be set at the beginning of each test block, **test_value** and **test_name**. In most cases, **test_value** will be a string that represents the error code that the test will return or the name of the **OK** function. **test_name** should be set to the appropriate string in the array of names in the **setup files** (indexed by the **test number**).

2.4.2 OK Functions

Each test (except **exs_init**) has prerequisite functions that need to be used to set up the test case. In each case, call these functions in the following manner:

```
if( ok_exs_init(p) < 0 ) exit(EXIT_FAILURE)
```

This ensures the prerequisite functions will terminate the test suite if they fail. If the prerequisite functions set or return any variables that are necessary in the test block, retrieve them through the **param struct**.

2.4.3 Using Param Struct

The **param struct** is passed to each test block through the variable **p**. This is a direct pointer to a **param struct** and thus the appropriate variables can be accessed directly. All parameters that will be passed to the function being tested should be retrieved from **p**.

2.4.2 Use of FIFOs

This section only applies to **blocking** mode tests. In some test cases it may be necessary to synchronize the execution of the test – i.e. setting up the server for listening before a client connects. This is done using **fifos** which are created when the **setup_options** function executes. Here is an example on how to use the **fifos**.

Thread 1:

```
int fd = open(exs_test_fifo_1,O_RDONLY);
/** executing second **/
close(fd)
/** execute after server **/
fd = open(exs_test_fifo_2, O_WRONLY);
close(fd)
```

Thread 2:

```
/** executing first **/
int fd = open(exs_test_fifo_1,O_RDONLY);
close(fd)
/** execute after client **/
fd = open(exs_test_fifo_2, O_WRONLY);
close(fd)
```

2.4.4 Function Call

The function to be tested should always be isolated on its own line and called using the appropriate **exs** function call. The **return value** of the function should always be stored in an **instance variable** and passed to the **print_macro** function to ensure the result of the test is accurate. For example:

```
int ret = exs_send(p->fd, p->send_buf, p->SEND_BUFSIZE, p->flags, p-
>qh, p->ah, p->mh);
```

Note how the **parameters** passed to the function are all obtained from **p**. This is best practice.

2.4.5 Using print_macro

The call to **print_macro** is crucial to ensure the framework reports accurate results. This method is explained in detail in **2.1.3.1** and **3**. The return value of the function call to be tested should always immediately be passed to **print_macro**.

2.4.6 Operation Mode

Some tests may behave differently or have different initialization steps when operating in **non-blocking** rather than **blocking** mode. For instance, **exs_qcreate** must be called in **non-blocking** mode but is unnecessary in **blocking** mode.

To determine the current mode of operation, check the value in **p** named **mode.** If this value is **1** then the framework is operating in **non-blocking** mode. Otherwise, it is **blocking** mode.

If a test is not applicable in a given mode, simply include the following line into the test block:

```
if( p->mode == 1 ) return 0;
```

This terminates test block if framework is in non-blocking mode.

2.4.7 Test Block Example

This is an example of a test block. This test is for **exs_send** and is testing whether or not the function will return the **ENOTCONN** error code if the **socket** specified has never been connected.

```
/* ENOTCONN: socket never connected */
if( test == 1 ){
      p->test_value = "ENOTCONN";
      p->test_name = all_send_tests[test-1];

      if( ok_exs_init(p) < 0 ) exit(EXIT_FAILURE);

      if( p->mode == 1 ) if( ok_exs_qcreate(p) < 0 )
            exit(EXIT_FAILURE);

      if( ok_exs_socket(p) < 0 ) exit(EXIT_FAILURE);

      int ret = exs_send(p->fd, p->send_buf, p->SEND_BUFSIZE,
            p->flags, p->qh, p->ah, p->mh);

      print_macro(test,p->test_value,p->test_name,ret,-1,
                  ENOTCONN);
}
```

3. Results

Test results will be printed by **print_macro** (and its variants) to **stdout** and also to a **log** file. A result will be printed on a single line in **7** separate parts, delimited by a **":"**. It will be sufficient to **diff** two result outputs to determine which tests changed (provided the same tests were run in the same order).

Test Number	This is the number used to specify the test. Specifying this number in the command line will run this test.
Test Value	This is the **expected** test value to be returned from the test. Either an error code, or an **OK** function name.
Test Name	This displays the parameters of the call, along with additional information to differentiate the tests.
Mode	This displays either **blocking** or **non-blocking** mode.
Result	This simply displays **PASS** or **FAIL**
Return Value	This is the value returned by the function, usually an error code or 0 on success.
Reason	This is the error code translated by **strerror**, or displays **SEGFAULT**, or **Time Out**

4. Modifying Existing Test Cases

To modify an existing test case, it is usually only necessary to edit the **test file** and/or **setup file**.

To modify an existing test, the appropriate test block needs to be modified. Be sure to modify the **print_macro** call if the **result** changes.

5. Adding Additional Test Cases

To add an additional test case to an **existing** function, it is usually only necessary to modify the **test file** and/or **setup file**.

Adding a new **test block** requires incrementing the **FUNCTION_TESTS** definition in the **setup** file and implementing a new test following the guidelines in **Test Blocks**.

To add a new **function** to the framework, creating a new **test file** and **setup** file is necessary, along with modifying **list_tests** in **result.h**. It is critical to utilize the **print_macro** function correctly to ensure results are properly displayed.

If a new **function** requires creating new function in **exs.c**, see **Section 7**.

6. Miscellaneous

Additional notes that are not applicable to mention on other sections of this document.

6.1 Scripts

It may be helpful to utilize **bash** scripts to help when running multiple tests in a manual or automated fashion. For example, here is an example of a script that will run through all supported tests in this framework and pass along parameters that will be passed to each function:

```
timeout 30s ./test_exs_init $@ 2>/dev/null
timeout 30s ./test_exs_socket $@ 2>/dev/null
timeout 30s ./test_exs_connect $@ 2>/dev/null
timeout 30s ./test_exs_blocking_connect $@ 2>/dev/null
timeout 30s ./test_exs_bind $@ 2>/dev/null
timeout 30s ./test_exs_listen $@ 2>/dev/null
timeout 30s ./test_exs_accept $@ 2>/dev/null
timeout 30s ./test_exs_blocking_accept $@ 2>/dev/null
timeout 30s ./test_exs_close $@ 2>/dev/null
timeout 30s ./test_exs_blocking_close $@ 2>/dev/null
timeout 30s ./test_exs_send $@ 2>/dev/null
timeout 30s ./test_exs_blocking_send $@ 2>/dev/null
timeout 30s ./test_exs_write $@ 2>/dev/null
timeout 30s ./test_exs_recv $@ 2>/dev/null
timeout 30s ./test_exs_blocking_recv $@ 2>/dev/null
timeout 30s ./test_exs_read $@ 2>/dev/null
timeout 30s ./test_exs_shutdown $@ 2>/dev/null
timeout 30s ./test_exs_mregister $@ 2>/dev/null
timeout 30s ./test_exs_mderegister $@ 2>/dev/null
timeout 30s ./test_exs_qcreate $@ 2>/dev/null
timeout 30s ./test_exs_qdelete $@ 2>/dev/null
timeout 30s ./test_exs_qdequeue $@ 2>/dev/null
timeout 30s ./test_exs_qstatus $@ 2>/dev/null
timeout 30s ./test_exs_qmodify $@ 2>/dev/null
timeout 30s ./test_exs_sendfile $@ 2>/dev/null
timeout 30s ./test_exs_blocking_sendfile $@ 2>/dev/null
```

Of course, because this framework outputs results to **stdout** the output can be piped to any other application or file.

7. Integrating into UNH EXS Build System

The testing framework has been integrated into the UNH EXS build system.

7.1 Framework Location

The testing framework is located in the **tests** directory in the base directory of the UNH EXS install.

7.2 Building the Framework

The framework is not built during make all install. Rather, the framework is only built if make check or make distcheck is run. This will compile all of the test cases in **tests** and run **test_all_check.sh** (which is defined in **7.3.4**). To run the tests manually, there are two options:

The first is to compile a test individually. For example,

```
make tests/test_exs_init
```

will build **test_exs_init** and it can be run as normal.

The second is to compile all tests but not run them. To do this, use the command

```
make TESTS= check
```

`make check` should produce the following output. The output of **test_all.sh** is found in **./test-suite.log**.

7.3 Changing UNH EXS

Brand new functions that are added to UNH EXS need to be forward declared in several locations in order to be built properly. The changes need to be made in the following locations:

7.3.1 exs.h

exs.h is located at **include/exs.h**. Find the following section:

```
/* Function Forward Declarations */
int        exs_init (int);
```

and add the new **function signature** here.

7.3.2 libexs.sym

libexs.sym is located at **libexs/libexs.sym**. The file starts with:

```
LIBEXS_1.2 {
global:
     exs_init;
```

and the new **function name** needs to be added to this list (with a **semicolon**).

7.3.3 Makefile.am

Two additions need to be made to **Makefile.am**.

The first is EXTRA_tests_test_exs_init_SOURCES = \. Add the test case include file to this list, e.g. tests/setup/exs_sendfile_setup.h.

The second is a new check_PROGRAMS addition. Add the follow lines (with exs_sendfile swapped out for the exs function you are adding:

```
check_PROGRAMS += tests/test_exs_sendfile
tests_test_exs_sendfile_SOURCES = tests/test_exs_sendfile.c
tests_test_exs_sendfile_CPPFLAGS = -I$(top_srcdir)/include
tests_test_exs_sendfile_LDADD = libexs/libexs.la
```

7.3.4 test_all_check.sh

This script does some error checking and then runs the script **test_all.sh** and pipes its output to **grep** which searches for the string **"FAIL"**. If it finds the string **test_all_check.sh** returns 1, else returns 0. This is done to integrate into the existing tests in the UNH EXS build system.

BIBLIOGRAPHY

[1] UNH Extended Sockets Library (UNH-EXS), https://www.iol.unh.edu/expertise/unh-exs

[2] The Open Group, *Extended Sockets API (ES-API) Issue 1.0*, The Open Group, ISBN: 1-931624-52-6, January 2005

[3] RFC 5040, *A Remote Direct Memory Access Protocol Specification*, https://tools.ietf.org/html/rfc5040

[4] POSIX, IEEE Std 1003.1-2008, 2016 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/

[5] socket(2), https://linux.die.net/man/2/socket

[6] close(2), https://linux.die.net/man/2/close

[7] OFED Overview, https://www.openfabrics.org/index.php/openfabrics-software.html

[8] UNH EXS 1.3.6, https://www.iol.unh.edu/sites/default/files/unh-exs/unh-exs-1.3.6.tar.gz

[9] mmap(2), munmap(2), https://linux.die.net/man/2/mmap

[10] UNH EXS 1.4.0, https://www.iol.unh.edu/sites/default/files/unh-exs/unh-exs-1.4.0.tar.gz