Consortium Specification

Extended Sockets API (ES-API)

Issue 1.0

The Interconnect Software Consortium

in association with

THE Open GROUP

Consortium Specification

Extended Sockets API (ES-API) Issue 1.0

ISBN: 1-931624-52-6

Document Number: C050

Published by The Open Group, January 2005.

Comments relating to the material contained in this document may be submitted to:

The Open Group, Thames Tower, 37-45 Station Road, Reading, Berkshire, RG1 1LX, U.K.

or by electronic mail to:

ogspecs@opengroup.org

Contents

Preface

The Interconnect Software Consortium

The purpose of the Interconnect Software Consortium is to develop and publish software specifications, guidelines, and compliance tests that enable the successful deployment of fast interconnects such as those defined by the InfiniBand specification.

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/testing.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/bookstore.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.
- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is the Consortium Specification for the Extended Sockets API. It has been developed and approved by The Interconnect Software Consortium in association with The Open Group.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, type names, and data structures
- *Italic* strings are used for emphasis. Italics in text also denote variable names and functions.
- Normal font is used for the names of constants and literals.
- Syntax and code examples are shown in `fixed width font`.
- ***Bold Italic*** is used for all terms defined in the Definitions section when they first appear in Chapter 1. IT-API objects are capitalized throughout the document (e.g., Interface Adapter, Endpoint, etc).

Trademarks

The Open Group® is a registered trademark of The Open Group in the United States and other countries

InfiniBand™ is a trademark of the InfiniBand Trade Association.

POSIX® is a registered trademark of the IEEE.

The Open Group acknowledges that there may be other brand, company, and product names used in this document that may be covered by trademark protection and advises the reader to verify them independently.

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following people in the development of this document:

Anthony Chen, Intel
Jerry Chu, Sun
David Edmondson, Sun
Annie Foong, Intel
Jeremy Harris, Sun
Al Hartmann, Vieo
Carl Hensler, Sun
Juliana Hsu, IBM
Masanori Itoh, Fujitsu
John Kasperski, IBM
Michael Krause, HP
Louis Laborde, HP
Jack McCann, HP
Jeffrey Messing, IBM
Ken Mintz, HP
Toshiaki Saeki, Fujitsu
Matthias Scheler, Sun
Satya Sharma, IBM
Vish Subramanian, HP
Robert Teisberg, HP
Mark Wittle, Network Appliance

Referenced Documents

The following document is referenced in this specification:

- Infiniband Architecture Release 1.1 Specification, Infiniband Trade Association

1 Introduction

The Socket API Extensions Working Group is working to provide extensions to the traditional socket API to support improved efficiency in network programming.

To that end, the group has developed an API specification that includes:

- Synchronous IO and control operations on sockets
- Event queue-based management of asynchronous operations
- Pre-registering of memory regions that will be the subject of IO operations

These facilities are intended to support:

- Improved efficiency when dealing with high numbers of socket file descriptors
- “Zero-copy” transmit and receive operations
- Improved buffer management

The Extended Socket API (ES-API) includes routines that provide asynchronous IO and control operations, asynchronous operation management, and memory registration functions for applications manipulating sockets.

The functions are divided into four main sections as follows:

- Asynchronous Operation Management functions

  The *exs_init*() function is used to declare that an application intends to use the facilities provided by the extended socket API. *exs_init*() must be called by an application before any other extended socket API function will succeed.

  exs_qcreate(), *exs_qdelete*(), *exs_qstatus*(), *exs_qmodify*(), and *exs_qdequeue*() allow applications to create, delete, examine the attributes of, modify the attributes of, and examine the contents of queues of event structures. The completion of all asynchronous operations is managed through such queues. When initiating any asynchronous operation, applications must indicate the queue where a completion event will be delivered.

  The *exs_cancel*() function attempts to cancel a previously initiated asynchronous operation. Cancelled operations result in completion events marked as cancelled.

- Asynchronous IO functions

  The *exs_send*(), *exs_sendmsg*(), *exs_recv*(), *exs_recvmsg*(), and *exs_sendfile*() functions provide support for initiating asynchronous IO operations. Socket descriptors, buffer pointers, and file descriptors are passed as arguments to the IO initiation functions,

together with a completion queue, a memory handle, and an application handle (see below).

- Asynchronous Control operations

  The *exs_connect*() function initiates an asynchronous connect operation on a socket. This includes indicating a default destination for sockets which are otherwise connectionless.

  The *exs_accept*() function allows an application to request that inbound connection notifications are delivered as events to a nominated queue. The events returned from the queue include the file descriptor of the new connection.

  exs_poll() is used to manage the set of socket descriptors that are monitored for activity in a manner similar to *poll*(2). Any activity on so monitored descriptors is reported to applications through events delivered to a nominated queue.

- Memory Management functions

  The *exs_mregister*(), *exs_mderegister*(), and *exs_mmodify*() functions allow applications to register regions of memory that may be used in future asynchronous IO operations. The system may use this registration to improve the performance of IO operations.

In general, the operation of the functions included in the API is intended to mirror their synchronous counterparts. There are some notable exceptions:

- Buffers referenced by an application when initiating IO operations are considered to be owned by the implementation until the IO operation completes. Applications must not attempt to access the contents of such buffers during an asynchronous operation as undefined behavior may result.

- Applications are encouraged to register buffers that will be used in asynchronous IO operations via the *exs_mregister*() function. This may provide the implementation with an opportunity to improve the performance of the IO operation. However, this registration is not required and, in such cases, the memory handle parameter **EXS_MHANDLE_UNREGISTERED** is used.

- The memory handles returned from *exs_mregister*() are intended to have meaning only in the context of the implementation. Applications should not interpret the contents of a memory handle and may not make assumptions based on comparisons or differences therein. Memory handles are not portable between multiple process contexts.

- Applications may provide an application handle when initiating asynchronous operations. This handle is intended to allow applications to track some element of application state with which the operation is associated. The type of the handle is such that a pointer to any object in the memory model of the application may be used. The implementation will not examine the contents of application handles and makes no assumptions about their form or meaning.

- The operations in the API are specified only when the target file descriptor refers to a socket. It is expected that the specified operations will operate appropriately should an implementation choose to support the operations on non-socket file descriptors, though that is outside the scope of this specification.

2 Header File

The Extended Socket API (ES-API) is supported by a single header file, **<sys/ex.h>**.

sys/exs.h

NAME

exs.h – extended socket header file

SYNOPSIS

#include <sys/exs.h>

DESCRIPTION

The **<sys/exs.h>** header file defines macros, types, and declares functions for the extended socket API.

The following macros are defined for use as the *version* argument in *exs_init*() calls:

EXS_VERSION /* Preferred version of the implementation. */

EXS_VERSION1 /* Initial version of the specification. */

The type **exs_ahandle_t** is defined to hold asynchronous handles which are identifiers that allow applications to track asynchronous operations. This type is large enough to hold an object pointer in the memory model of the application. The implementation will not attempt to interpret the value of **exs_ahandle_t** in any way. The implementation will return an undefined value for **exs_ahandle_t**, when none is provided by an application, such as when the function *exs_accept*() is called without an **exs_acceptaddr_t** array.

The following macro is defined for use as the *flags* argument in *exs_mregister*() and *exs_mmodify*() calls:

EXS_MRF_SHARED

The type **exs_mhandle_t** is defined to hold registered memory handles allocated by calling the *exs_mregister*() function. Applications should not attempt to interpret the value of any **exs_mhandle_t** in any way. The following macros:

EXS_MHANDLE_INVALID
EXS_MHANDLE_UNREGISTERED

are also defined as special distinct values of **exs_mhandle_t** to indicate an invalid handle and a handle for a non-registered memory region, respectively.

The type **exs_qhandle_t** is defined to hold event queue handles allocated by calling the *exs_qcreate*() function. Applications should not attempt to interpret the value of any **exs_qhandle_t** in any way. The macro **EXS_QHANDLE_INVALID** is also defined as a special value of **exs_qhandle_t** to indicate an invalid event queue handle.

The following macros are defined with distinct integer values, for use as the *attr_type* argument in *exs_qstatus*() and *exs_qmodify*() calls:

EXS_QATTR_DEPTH
EXS_QATTR_SIGNAL
EXS_QATTR_EVENTS

The structure **exs_signal** and the type **exs_signal_t** are defined for use in *exs_qmodify()* and *exs_qstatus()* calls and include at least the following members:

```
int exs_sigstate
int exs_signo
```

The following constants shall be defined as distinct integer values for the *exs_sigstate* member in **exs_signal_t***:*

EXS_SIG_ENABLE
EXS_SIG_DISABLE

The structure **exs_pollfd** and the type **exs_pollfd_t** are defined for use in *exs_poll()* calls and include at least the following members:

```
int           exs_fildes    /* File descriptor. */
int           exs_events    /* Input event flags. */
exs_ahandle_t exs_ahandle   /* Asynchronous handle. */
```

The following macros are defined with bitwise-distinct values, to form the *exs_events* member in **exs_pollfd_t**:

EXS_POLLIN
EXS_POLLOUT

The structure **exs_acceptaddr** and the type **exs_acceptaddr_t** are defined for use in *exs_accept()* calls and include at least the following members:

```
struct sockaddr  *exs_addr      /* Optional address. */
socklen_t         exs_addrlen   /* Size of address. */
exs_ahandle_t     exs_ahandle
```

The structure **exs_iovec** and the type **exs_iovec_t** are defined and include at least the following members:

```
void         *iov_base    /* Base address of a memory region. */
size_t        iov_len     /* Size of the memory pointed to
                             by iov_base. */
exs_mhandle_t iov_mhandle /* mhandle tied to the memory at iov_base. */
```

The structure **exs_msghdr** and the type **exs_msghdr_t** are defined for use in *exs_sendmsg()* and *exs_recvmsg()* calls and include at least the following members:

```
void         *msg_name        /* Optional address. */
socklen_t     msg_namelen     /* Size of address. */
exs_iovec_t  *msg_iov         /* Scatter/gather array. */
int           msg_iovlen      /* Members in exs_msg_iov. */
void         *msg_control     /* Ancillary data. */
socklen_t     msg_controllen  /* Ancillary data buffer length. */
int           msg_flags       /* Flags on received message. */
```

The structure **exs_fdvec** and the type **exs_fdvec_t** are defined and include at least the following members:

```
int     exs_fildes    /* File descriptor.*/
off_t   exs_offset    /* Offset into the file. */
size_t  exs_length    /* Requested transfer length. */
```

```
int     exs_flags    /* Flags. */
```

The structure **exs_xferfile** and the type **exs_xferfile_t** are defined for use in *exs_sendfile*() calls and include at least the following members:

```
int  exs_xfv_type           /* Source extent type. */
union {
    exs_iovec_t  exs_iov    /* IOVEC extent. */
    exs_fdvec_t  exs_fdv    /* FDVEC extent. */
} exs_xfv                   /* Source extent. */
```

The following macros are defined with distinct integer values, for use in the *exs_xfv_type* member of **exs_xferfile_t**:

EXS_IOVEC
EXS_FDVEC

The following macro is defined with value bitwise-distinct from other possible values, for use in the *flags* arguments in *exs_sendfile*() calls:

EXS_SHUT_WR

The following macros are defined with integer values distinct from other **SOL_SOCKET** socket-level options defined in **<sys/socket.h>**, for use as the *option_name* argument in *getsockopt*() and *setsockopt*() calls:

SO_ASYNC_RECV_ORDERED
SO_ASYNC_SEND_TIMEOUT
SO_ASYNC_RECV_TIMEOUT

The **SO_ASYNC_RECV_ORDERED** option is used to control the order in which buffers, passed to the implementation using *exs_recv*() or *exs_recvmsg*(), are returned in events. This option takes an **int** value. This is a Boolean option. When **SO_ASYNC_RECV_ORDERED** is set, the implementation must return the buffers in the order they were posted. When **SO_ASYNC_RECV_ORDERED** is not set, the implementation is free to return the buffers in any order. By default, **SO_ASYNC_RECV_ORDERED** is not set.

SO_ASYNC_SEND_TIMEOUT and **SO_ASYNC_RECV_TIMEOUT** specify time limits for completing send and receive operations. If the time limit is reached, a timeout event is delivered to the specified event queue, but any outstanding send or receive operations are not cancelled or failed.

Both options expect a pointer to an **exs_timeout_t** structure to define how the timeout should occur. **exs_timeout_t** includes at least the following members:

```
struct timeval exs_duration;   /* Timeout value. */
int exs_flags;                 /* Flags specifying timer
                                  restart behavior (see below). */
exs_qhandle_t  exs_qhandle;    /* Queue to which the events */
                                  will be delivered. */
exs_ahandle_t  exs_ahandle;    /* Application supplied handle. */
```

exs_flags determines whether or not the timer is automatically restarted once a timeout has occurred. By default, the timer is automatically restarted. Applications may set the *exs_flags* value to **EXS_TIMEOUT_ONCE** to prevent the automatic restart of the timer.

Setting the *exs_duration* member of the **exs_timeout_t** structure to zero will disable the timer. The timer is disabled by default.

exs_qhandle is the event queue to which any of these timeout events will be delivered. This event queue need not be the same event queue as that indicated in the original asynchronous IO operation.

If a timeout value is specified by an application, the timer will run whenever there are pending operations of the appropriate type. For **SO_ASYNC_SEND_TIMEOUT,** appropriate operations are those initiated by the following functions:

exs_send()
exs_sendmsg()
exs_sendfile()

For **SO_ASYNC_RECV_TIMEOUT,** appropriate operations are those initiated by the following functions:

exs_recv()
exs_recvmsg()

The timer value is reset to its maximum when an appropriate completion event is delivered to the corresponding event queue and there are pending operations; otherwise, the timer is stopped. The timer will restart when the next appropriate operation is initiated.

A timeout event will be delivered to the event queue indicated in the **exs_timeout_t** structure if the timer value reaches zero. The following fields of the **exs_event_t** structure that represents the timeout will contain relevant data:

exs_evt_type	Set to EXS_EVT_SENDTIMEOUT or EXS_EVT_RECVTIMEOUT, as appropriate.
exs_evt_errno	Set to ETIMEDOUT.
exs_evt_ahandle	Set to the exs_ahandle value from exs_timeout_t.
exs_evt_socket	Set to the socket.

If **EXS_TIMEOUT_ONCE** was not set in the *exs_flags* member of the **exs_timeout_t** structure, the timer will automatically be reset to its maximum and will continue to run after the timeout event has been delivered and there are pending operations; otherwise, it is reset on the next appropriate operation.

If **EXS_TIMEOUT_ONCE** was set in the *exs_flags* member of the **exs_timeout_t** structure, the timer will be disabled after delivery of the timeout event. It can be re-enabled by a subsequent call to *setsockopt*().

The structure **exs_event** and the type **exs_event_t** are defined for use in *exs_qdequeue*() calls, and include at least the following members:

```
int           exs_evt_type
int           exs_evt_errno   /* Status: 0 means success. */
exs_ahandle_t exs_evt_ahandle /* Set to value passed by application. */
int             exs_evt_socket
union exs_evt_u exs_evt_union /* Union of event-specific structures. */
```

The union *exs_evt_u* is defined as an element of the type **exs_event_t**. The union's members contain the elements specific to different event types. They include at least the following members:

```
exs_evt_poll_t       exs_evt_poll
exs_evt_accept_t     exs_evt_accept
exs_evt_xfer_t       exs_evt_xfer
exs_evt_xfermsg_t    exs_evt_xfermsg
exs_evt_sendfile_t   exs_evt_sendfile
```

The following macros are defined with distinct integer values, for use as the *exs_evt_type* member of **exs_event_t**:

EXS_EVT_POLL
EXS_EVT_CONNECT
EXS_EVT_ACCEPT
EXS_EVT_SEND
EXS_EVT_RECV
EXS_EVT_SENDMSG
EXS_EVT_RECVMSG
EXS_EVT_SENDFILE
EXS_EVT_SENDTIMEOUT
EXS_EVT_RECVTIMEOUT

The following macro is defined for the purpose of defining the maximum number of events that can be returned in a call to the *exs_qdequeue*() routine.

EXS_EVTVEC_MAX

The structure **exs_evt_poll** and the type **exs_evt_poll_t** are defined. They include the specific elements for the event associated with the *exs_poll*() function and include at least the following member:

```
int  exs_evt_events
```

The structure **exs_evt_accept** and the type **exs_evt_accept_t** are defined. They include the specific elements for the event associated with the *exs_accept*() function and include at least the following members:

```
int              exs_evt_new_socket
struct sockaddr *exs_evt_addr
socklen_t        exs_evt_addrlen
```

The structure **exs_evt_xfer** and the type **exs_evt_xfer_t** are defined. They include the specific elements for the event associated with the *exs_send*() and *exs_recv*() functions and include at least the following members:

```
void           *exs_evt_buffer
size_t          exs_evt_length    /* Number of bytes transferred. */
exs_mhandle_t   exs_evt_mhandle
```

The structure **exs_evt_xfermsg** and the type **exs_evt_xfermsg_t** are defined. They include the specific elements for the event associated with the *exs_sendmsg*() and *exs_recvmsg*() functions and that includes at least the following members:

```
exs_msghdr_t  *exs_evt_msg
size_t         exs_evt_length    /* Number of bytes transferred. */
```

The structure **exs_evt_sendfile** and the type **exs_evt_sendfile_***t* are defined. They include the specific elements for the event associated with the *exs_sendfile*() function and that includes at least the following members:

```
exs_xferfile_t *exs_evt_sendvec
int             exs_evt_sendvec_cnt
size_t          exs_evt_length       /* Number of bytes transferred. */
```

The following macros are defined with distinct integer values, for use as the *flags* argument in *exs_cancel*() calls:

EXS_CAF_FILDES
EXS_CAF_AHANDLE

The following are declared as functions and may also be defined as macros. Function prototypes will be provided.

```
int            exs_init (int);
exs_mhandle_t  exs_mregister (void*, size_t, int);
int            exs_mmodify (exs_mhandle_t, size_t, int);
int            exs_mderegister (exs_mhandle_t, int);
exs_qhandle_t  exs_qcreate(int);
int            exs_qdelete (exs_qhandle_t);
int            exs_qmodify (exs_qhandle_t, int, void*, size_t);
int            exs_qdequeue (exs_qhandle_t, exs_event_t*, int,
                   const struct timeval*);
int            exs_qstatus (exs_qhandle_t, int, void*, size_t);
int            exs_cancel (int, int, exs_ahandle_t);
int            exs_accept (int, exs_acceptaddr_t*, int, int,
                   exs_qhandle_t);
int            exs_connect (int, const struct sockaddr*, socklen_t,
                   int, struct timeval*, exs_qhandle_t, exs_ahandle_t);
ssize_t        exs_send (int, const void*, size_t, int, exs_qhandle_t,
                   exs_ahandle_t, exs_mhandle_t);
ssize_t        exs_sendmsg (int, const exs_msghdr_t*, int,
                   exs_qhandle_t, exs_ahandle_t);
ssize_t        exs_recv (int, void*, size_t, int, exs_qhandle_t,
                   exs_ahandle_t, exs_mhandle_t);
ssize_t        exs_recvmsg (int, exs_msghdr_t*, int, exs_qhandle_t,
                   exs_ahandle_t);
ssize_t        exs_sendfile (int, exs_xferfile_t*, int, int,
                   exs_qhandle_t, exs_ahandle_t);
int            exs_poll (exs_pollfd_t*, nfds_t, int, exs_qhandle_t);
```

3 Reference Pages

The Extended Sockets API (ES-API) consists of the following interfaces:

exs_accept	Asynchronously accept incoming connections on a socket.
exs_cancel	Cancel pending asynchronous operations.
exs_connect	Asynchronously connect a socket.
exs_init	Extended socket API initialization.
exs_mderegister	Deregister application memory.
exs_mmodify	Modify registered application memory.
exs_mregister	Register application memory.
exs_poll	Modify the set of monitored conditions for multiple sockets to trigger asynchronous notification events on a specified event.
exs_qcreate	Create an event queue.
exs_qdelete	Delete an event queue.
exs_qdequeue	Retrieve events from an event queue.
exs_qmodify	Modify event queue attributes.
exs_qstatus	Retrieve event queue attributes.
exs_recv	Asynchronously receive a message from a connected socket.
exs_recvmsg	Asynchronously receive a message from a socket.
exs_send	Asynchronously send a message on a socket.
exs_sendfile	Initiate transmission of the contents of a file from a socket.
exs_sendmsg	Asynchronously send a message on a socket.

exs_accept()

NAME

exs_accept – asynchronously accept incoming connections on a socket

SYNOPSIS

```
#include <sys/exs.h>

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

DESCRIPTION

The *exs_accept*() function extracts *addrvec_cnt* connections from the queue of pending connections, creates new sockets with the same socket type, protocol, and address family as the specified socket, and allocates a new file descriptor for each of these sockets.

The *exs_accept*() function takes the following arguments:

socket Specifies a socket that was created with *socket*(), has been bound to an address with *bind*(), and has issued a successful call to *listen*().

addrvec Points to an array of **exs_acceptaddr** structures. This structure is defined in the **<sys/exs.h>** header file and contains at least the following members:

```
struct sockaddr   *exs_addr
socklen_t          exs_addrlen
exs_ahandle_t      exs_ahandle
```

Each of these structures contains a pointer to **sockaddr** structure that will be updated to contain the peer addresses of the incoming connection. A null pointer may be specified for the *addrvec* argument if the application does not want the peer address of the incoming connection to be returned. If the *addrvec* argument is not null, the array must contain *addrvec_cnt* elements.

The *exs_ahandle* field specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request. In the case where the *addrvec* is NULL, the *exs_ahandle* field in the event structure is undefined. The application may use this value to identify this request after retrieving the completion event by calling *exs_qdequeue*() or when calling *exs_cancel*().

The *exs_addr* field may be null, if the caller wishes to specify the *exs_ahandle* field, but does not want the peer address of the incoming connection to be returned.

addrvec_cnt Specifies the number of connections that should be accepted asynchronously for the specified listening socket before the application needs to call *exs_accept*()

again. If *addrvec* is not a null pointer, *addrvec_cnt* must also be the number of **exs_acceptaddr** structures in the array pointed to by the *addrvec* argument.

flags Specifies how the *exs_accept*() operation should behave. There are currently no values defined and this argument should be set to 0.

qhandle Specifies the destination event queue on which the completion event should be posted. This event queue must have previously been created using *exs_qcreate*().

If *addrvec* is not a null pointer, the address of the peer for the accepted connection is stored in the **sockaddr** structure pointed to by the *exs_addr* field in the **exs_sockaddr** structure. If the actual length of the peer address is greater than the length of the supplied **sockaddr** structure as indicated by the *exs_addrlen* field in the **exs_acceptaddr** structure, the stored address will be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the buffer pointed to by the *exs_addr* argument is unspecified.

When a connection becomes available at the listener socket, an event will be enqueued on the specified event queue. The completion event can be retrieved from the event queue with the *exs_qdequeue*() function.

The completion event for the *exs_accept*() operation is the **exs_event** structure. This structure is defined in the **<sys/exs.h>** header file and it contains at least the following members:

int exs_evt_type Set to **EXS_EVT_ACCEPT**.

int exs_evt_errno Set to indicate the status of the I/O operation.

exs_ahandle_t exs_evt_ahandle
Set to the *exs_ahandle* value that corresponds to the *exs_addr* address that is used to set *exs_evt_addr*.

int exs_evt_socket Set to the value of the *socket* argument.

union exs_evt_u exs_evt_union
The *exs_evt_union* field contains the **exs_evt_accept** structure that contains at least the following members:

int exs_evt_new_socket
Set to the file descriptor associated with the incoming connection, if the operation was successful.

*struct sockaddr *exs_evt_addr*
Set to the memory location where the peer address of the incoming connection is stored, if the operation was successful. The address in *exs_evt_addr* is one of the *exs_addr* addresses passed in *addrvec*.

socklen_t exs_evt_addrlen
Set to the length of the peer address that is stored in the memory location pointed to by *exs_evt_addr*.

Possible values that could be returned in the *exs_evt_errno* field include:

0	The asynchronous accept operation was successful.
[ECANCELED]	The operation has been cancelled.
[EFAULT]	The system detected an address that was not valid while attempting to access the *exs_addr* argument.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum number of file descriptors in the system is already open.

The accepted socket, *exs_evt_new_socket*, is used to read and write data to and from the connection. It is not used to accept more connections. The original listening socket, *exs_evt_socket*, remains open for accepting subsequent incoming connections.

The application should not access the memory buffer that is described by the *exs_addr* argument until the completion event is retrieved by calling *exs_qdequeue()*.

If the socket referenced by the *socket* argument is closed before *addrvec_cnt* incoming connections have been accepted, a completion event for each outstanding operation will be delivered to the event queue specified by the *qhandle* argument indicating the failure.

The *exs_accept*() function will not be affected by the **O_NONBLOCK** flag.

Multiple *exs_accept*() operations will be serviced in implementation-defined order. If *addrvec_cnt* is larger than one, the *addrvec* array elements from a single *exs_accept*() operation will be returned in an implementation-defined order.

RETURN VALUE

The *exs_accept*() function will return one of the following values:

0	The *exs_accept*() operation has been initiated. Completion events will be posted to the event queue that was specified by the argument *qhandle*.
-1	The *exs_accept*() operation has failed. *errno* is set to indicate the error.

ERRORS

The *exs_accept*() function **will** fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNABORTED]	The *exs_accept*() function was issued on a socket for which receives have been disallowed (due to a *shutdown*() call).
[EFAULT]	The system detected an address that was not valid while attempting to access the *addrvec* or *exs_addr* argument.
[EINVAL]	The *qhandle* is not a valid event queue, the *socket* is not accepting connections (a *listen*() has not been issued), or a value of the *addrvec_cnt* <= 0.

[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The socket type of the specified socket does not support accepting connections.
[EPERM]	The *exs_init*() function must be called first.

The *exs_accept*() function **may** fail if:

[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	There was insufficient memory available to complete the operation.

SEE ALSO

exs_cancel(), *exs_init*(), *exs_qcreate*(), *exs_qdequeue*()

exs_cancel()

NAME

exs_cancel – cancel pending asynchronous operations

SYNOPSIS

```
#include <sys/exs.h>

int exs_cancel (int            flags,
                int            fildes,
                exs_ahandle_t  ahandle)
```

DESCRIPTION

The function *exs_cancel*() can be used to attempt to cancel any outstanding asynchronous operation.

flags Set to either **EXS_CAF_AHANDLE** or **EXS_CAF_FILDES**. The *flag* field determines how the *exs_cancel*() interface will select the operations to cancel.

fildes Specifies the socket to cancel operations on.

ahandle Specifies the correlator of the operations to cancel.

If flag is set to **EXS_CAF_AHANDLE**, then an attempt will be made to cancel all operations associated with the application handle specified by *ahandle*.

If flag is set to **EXS_CAF_FILDES**, an attempt will be made to cancel all outstanding operations initiated on the file descriptor specified by *fildes*.

In neither of these cases will an attempt be made to cancel existing timer activity (see **SO_ASYNC_SEND_TIMEOUT** and **SO_ASYNC_RECV_TIMEOUT**) or change the set of monitored conditions managed using *exs_poll*().

Only one of **EXS_CAF_AHANDLE** or **EXS_CAF_FILDES** may be specified. In each case the alternate function argument is ignored.

Cancel may not be possible for all operations. In any case the *exs_cancel*() function will return success if an attempt to cancel the specified operation(s) has been made. If an operation is successfully cancelled, its completion will appear in the event queue with the event's *errno* field set to **ECANCELED**. However, the cancel may not be possible, or the operation may have already completed, so an operation may complete normally or in error after *exs_cancel*() has been initiated.

If an *exs_send*(), *exs_sendmsg*(), or *exs_sendfile*() is canceled successfully, the user is guaranteed that the sent data will not have been received by the peer.

RETURN VALUE

If the operation is successful, *exs_cancel*() returns 0. Otherwise, *exs_cancel*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EBADF] Not a valid file descriptor.

[EINVAL]	No outstanding operations associated with the *ahandle* or file descriptor.
[ENOTSOCK]	The *fildest* argument does not refer to a socket.
[EPERM]	The *exs_init*() function must be called first.

SEE ALSO

exs_init(), *exs_poll*()

exs_connect()

NAME

exs_connect – asynchronously connect a socket

SYNOPSIS

```
#include <sys/exs.h>

int exs_connect(int                    socket,
                const struct sockaddr *address,
                socklen_t              address_len,
                int                    flags,
                const struct timeval  *timeout,
                exs_qhandle_t          qhandle,
                exs_ahandle_t          ahandle)
```

DESCRIPTION

The *exs_connect*() function initiates an asynchronous connect operation on a socket. The *exs_connect*() function takes the following arguments:

socket Specifies the file descriptor associated with the socket.

address Points to a **sockaddr** structure containing the peer address. The length and format of the address depend on the address family of the socket.

address_len Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

flags Specifies how the *exs_connect*() operation should behave. There are currently no values defined and this argument should be set to 0.

timeout Specifies how long the asynchronous connect operation should take before timing out. If the *timeout* argument is a null pointer, then the default timeout value associated with the *connect*() function will be used.

qhandle Specifies the destination event queue on which the completion event should be posted. This event queue must have previously been created using *exs_qcreate*().

ahandle Specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request. The application may use this value to identify this request after retrieving the completion event by calling *exs_qdequeue*() or when calling *exs_cancel*().

If the socket has not already been bound to a local address, *exs_connect*() will bind it to an address which, unless the socket's address family is **AF_UNIX**, is an unused local address.

If the initiating socket is not connection-mode, then *exs_connect*() will set the socket's peer address, and no connection is made. For **SOCK_DGRAM** sockets, the peer address identifies where all datagrams are sent on subsequent *send*() functions, and limits the remote sender for subsequent *recv*() functions. If *address* is a null address for the protocol, the socket's peer address will be reset. The *timeout* argument is ignored for **SOCK_DGRAM** sockets. Once the

socket's peer address has been set or reset, a completion event will be placed on the event queue that was specified by the *qhandle* argument.

If the initiating socket is connection-mode, then *exs_connect*() will attempt to establish a connection to the address specified by the *address* argument. Once the connection has been established, a completion event will be placed on the event queue that was specified by the *qhandle* argument. If the connection cannot be established within the length of time that was specified by the *timeout* argument, then a completion event will be posted to the specified event queue and it will have the *exs_evt_errno* field set to **ETIMEDOUT**. If an error is detected by the underlying protocol while attempting to establish the connection, a completion event will be posted to the specified event queue and the *exs_evt_errno* field will be set.

Subsequent calls to *connect*() or *exs_connect*() for the same socket, before the connection is established, will fail and set *errno* to **EALREADY**.

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. This structure is defined in the **<sys/exs.h>** header file and contains at least the following members:

int exs_evt_type Set to **EXS_EVT_CONNECT.**

int exs_evt_errno Set to indicate the status of the I/O operation.

exs_ahandle_t exs_evt_ahandle
Set to the value of the *ahandle* argument.

int exs_evt_socket Set to the value of the *socket* argument.

Possible values that could be returned in the *exs_evt_errno* field include:

0 The asynchronous connect operation was successful.

[ECANCELED] The operation has been cancelled.

[ECONNREFUSED] The target address was not listening for connections or refused the connection request.

[ECONNRESET] Remote host reset the connection request.

[EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ETIMEDOUT] The attempt to connect timed out before a connection was made.

If the socket referenced by the *socket* argument is closed before the connect operation is completed, an event will be delivered to the event queue specified by the *qhandle* argument indicating the failure.

The *exs_connect*() function will not be affected by the **O_NONBLOCK** flag.

The use of a timeout does not affect any pending timers set by *alarm*(), *ualarm*(), or *setitimer*().

The socket in use may require the process to have appropriate privileges to use the *exs_connect*() function.

RETURN VALUE

The *exs_connect*() will return one of the following values:

0	The *exs_connect*() operation has been initiated. A completion event will be posted to the event queue that was specified by the argument *qhandle*.
-1	The *exs_connect*() operation has failed. *errno* is set to indicate the error.

ERRORS

The *exs_connect*() function **will** fail if:

[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
[EALREADY]	A connection request is already in progress for the specified socket.
[EBADF]	The *socket* argument is not a valid file descriptor.
[EFAULT]	The system detected an address that was not valid while attempting to access the *address* or the *timeout* arguments.
[EINVAL]	The *qhandle* is not a valid event queue, the *address_len* is not a valid length for the address family, or there is an invalid address family in the **sockaddr** structure.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EISCONN]	The specified socket is connection-mode and is already connected.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in *address*.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters, or the pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
[ENETUNREACH]	No route to the network is present.
[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
[ENOTDIR]	A component of the path prefix of the pathname in *address* is not a directory.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EPERM]	The *exs_init*() function must be called first.

[EPROTOTYPE] The specified address has a different type than the socket bound to the specified peer address.

The *exs_connect*() function **may** fail if:

[EACCES] Search permission is denied for a component of the path prefix; or write access to the named socket is denied.

[EADDRINUSE] Attempt to establish a connection that uses addresses that are already in use.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[EOPNOTSUPP] The socket is listening and cannot be connected.

SEE ALSO

exs_cancel(), *exs_init*(), *exs_qcreate*(), *exs_qdequeue*()

exs_init()

NAME

exs_init – extended socket API initialization

SYNOPSIS

```
#include <sys/exs.h>

int  exs_init (int     version)
```

DESCRIPTION

The *exs_init*() function must be called prior to calling any other *exs_** programming interface function. This function must be called once per process. **EXS_VERSION** specifies the implementation's preferred version and is defined in **<sys/exs.h>**.

The *exs_init*() function takes the following arguments:

version Requested version of the API to be initialized.

RETURN VALUE

If the operation is successful, *exs_init*() returns 0. Otherwise, *exs_init*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EAGAIN]	Allocation of internal resources failed but a subsequent request may succeed.
[EALREADY]	*exs_init*() already successfully called.
[ENOTSUP]	The requested version is not supported.

RATIONALE

The primary purpose of the *exs_init*() function is to provide a means of supporting updated versions of the API while maintaining backwards compatibility.

SEE ALSO

None.

exs_mderegister()

NAME

exs_mderegister – deregister application memory

SYNOPSIS

```
#include <sys/exs.h>

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

DESCRIPTION

The *exs_mderegister*() function will deregister previously registered application memory. This routine may fail if all or a part of the memory in question has outstanding operations. If the routine succeeds and there are outstanding operations, the behavior of those operations is undefined. The caller should ensure that those operations are successfully completed or cancelled before the memory can be deregistered. This may require that associated sockets be closed.

The *exs_mderegister*() function takes the following arguments:

mhandle Specifies the handle to the memory being deregistered. This field was returned when it was registered.

flags Specifies options with regard to the deregistration. There are currently no values defined and this argument should be set to 0.

Each *mhandle* must be deregistered.

RETURN VALUE

If the operation is initiated successfully, *exs_mderegister*() returns 0. Otherwise, *exs_mderegister*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EALREADY] Operation already in-progress.

[EBUSY] There are still asynchronous operations outstanding.

[EINVAL] Invalid registered memory handle.

[EPERM] *exs_init*() must be called first.

SEE ALSO

exs_init()

exs_mmodify()

NAME

exs_mmodify – modify registered application memory

SYNOPSIS

```
#include <sys/exs.h>

int exs_mmodify(exs_mhandle_t  mhandle,
                size_t         size,
                int            flags)
```

DESCRIPTION

The *exs_mmodify*() function modifies an application's memory registration. The implementation will modify the length associated with the already registered buffer. This will allow applications to continue to use the referenced handle while adjusting the amount of referred memory.

The *exs_mmodify*() function takes the following arguments:

mhandle Memory handle being modified.

size New length of the application memory to be registered. The *size* must be >0.

flags Options for the registration (values defined in **<sys/exs.h>**):

EXS_MRF_SHARED Indicates that the application memory being registered is shared memory so that the implementation might optimize the registration. Use of the flag is optional.

This operation can increase and decrease the amount of memory registered by passing in a new length. In the event that the new size is smaller than the original size, the operation will not block waiting on outstanding operations using the memory being deregistered. Also, the existence of any outstanding operations which reference the memory region which is to be released by the *exs_mmodify*() call will cause *exs_mmodify*() to fail. This operation will not interfere with any outstanding operation depending on the memory registration.

In the case of an error, the original registration will remain valid.

RETURN VALUE

If the operation is successful, *exs_mmodify*() returns 0. Otherwise, *exs_mmodify*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EACCES] Permission denied for address range.

[EAGAIN] Allocation of internal data structures failed but a subsequent request may succeed.

[EBUSY] Memory being released is currently referenced by an in-progress operation.

[EFAULT] Bad address range.

[EINVAL]	Invalid *mhandle* or size.
[EPERM]	The *exs_init*() function was not called.

SEE ALSO

exs_init()

exs_mregister()

NAME

exs_mregister – register application memory

SYNOPSIS

```
#include <sys/exs.h>

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

DESCRIPTION

The *exs_mregister*() function registers the application memory in preparation for use in subsequent *exs_**() send and receive operations. The implementation will prepare the memory to be used for data transfers. The intention is that the implementation will use this registration to increase the performance of subsequent data transfers using the registered memory.

The *exs_mregister*() function takes the following arguments:

addr Address to the application memory to be registered.

size Length of the application memory to be registered. The size must be >0.

flags Options for the registration (values defined in **<sys/exs.h>**):

EXS_MRF_SHARED Indicates that the application memory being registered is shared memory so that the implementation might optimize the registration. Use of the flag is optional.

Following registration the application can use all or a portion of the registered memory for data transfers. Memory can be registered multiple times in whole or part. Note that each registration should be deregistered on termination.

Registered memory should not be freed without prior deregistration (see *exs_mderegister()*). Memory used in an asynchronous operation should not be modified while the operation is still outstanding.

Any data transfer routine which expects an **exs_mhandle_t** field, will accept **EXS_MHANDLE_UNREGISTERED** in its place. This will allow the routine to use unregistered memory. In this case, the user's buffer should not be accessed until the operation has been completed.

Any specified memory handle other than **EXS_MHANDLE_UNREGISTERED** must be a valid **exs_mhandle_t** type returned from this routine that has not already been de-registered using *exs_mderegister()*.

Registration is process scoped. Each process must register its own memory. Following a *fork* or *exec*, the child process would need to register its copy of any memory that will be used in future IO operations, as memory registration is not inherited by child processes.

RETURN VALUE

If the operation is successful, *exs_mregister*() returns a handle for the newly registered memory. Otherwise, *exs_mregister*() returns **EXS_MHANDLE_INVALID**, and *errno* is set to indicate the error.

ERRORS

[EACCES]	Permission denied for address range.
[EAGAIN]	Allocation of internal data structures failed but a subsequent request may succeed.
[EFAULT]	Bad address range.
[EINVAL]	Invalid flags or size.
[EPERM]	The *exs_init*() function must be called first.

SEE ALSO

exs_init(), *exs_mderegister*()

exs_poll()

NAME

exs_poll – modify the set of monitored conditions for multiple sockets to trigger asynchronous notification events on a specified event

SYNOPSIS

```
#include <sys/exs.h>

nfds_t exs_poll(exs_pollfd_t   fds[],
                nfds_t         nfds,
                int            flags,
                exs_qhandle_t  qhandle)
```

DESCRIPTION

The *exs_poll*() function takes the following arguments:

fds Specifies the address of an array of **exs_pollfd_t** structures.

nfds Specifies the number of **exs_pollfd_t** structure elements in the *fds* array.

flags Specifies the operation options. There are currently no values defined and this argument should be set to 0.

qhandle Specifies an event queue handle previously obtained with a call to *exs_qcreate*().

The **exs_pollfd_t** structure contains at least the following members:

int exs_fildes Specifies the socket descriptor being polled.

int exs_events Specifies the conditions to be monitored.

exs_ahandle_t exs_ahandle
Specifies the asynchronous handle that will be returned in the events triggered by this call.

The *exs_events* field indicates the requested conditions to be monitored, also called the "condition set" (abbreviated term for the phrase "set of interested conditions"), and is formed by OR'ing together zero or more of the following symbolic constants:

EXS_POLLIN Requests notification when data is available to be read from the socket or, in the case of a listening socket, a new inbound connection request is present.

EXS_POLLOUT Requests notification when data can be written to the socket.

Each element of the *fds* array is processed individually until all elements are processed or an error occurs. The *exs_poll*() function returns the number of consecutive array elements successfully processed. If an error occurs while processing a given element, none of the following elements, if any, will be processed.

The *exs_poll*() function is used to register or deregister a condition set for each socket and trigger asynchronous notification events on the specified event queue.

Registering is the process of creating or modifying an association between a socket, a condition set, and an event queue. Registering is accomplished by passing an *exs_events* field with a non-empty condition set; i.e., an *exs_events* field with a value different from 0. This registration replaces any previous registration for this socket on this event queue.

Deregistration is the process of removing an association between a socket, any condition set, and an event queue. Deregistration is accomplished by passing an *exs_events* field with an empty condition set; i.e., an *exs_events* field with a value of 0. Deregistration also happens implicitly when the socket is closed or when some internal error occurs which causes notification of an **exs_evt_poll_t** event with an *exs_evt_errno* status field different from 0. Deregistering by calling the *exs_poll*() function does not generate an **exs_evt_poll_t** event.

The active set of socket descriptors being monitored at any given time on a specific event queue is updated by the *exs_poll*() function in the following manner:

- A socket descriptor absent from the set will be added to the set if a registration is requested.
- A socket descriptor present in the set will have its registration updated if a registration is requested.
- A socket descriptor in the set will be removed from the set if a deregistration is requested.

The deregistration of a socket descriptor not present in the set will be silently ignored and still counted as a processed entry of the *fds* array in the return value of the *exs_poll*() function.

Once an active set of socket descriptors has been created or modified by one or more calls to *exs_poll*(), the application is notified of updated conditions through events delivered by the implementation to the specified event queue and retrieved by the application through calls to *exs_qdequeue*(). The events received are **exs_event** structures containing at least the following members:

int exs_evt_type Set to **EXS_EVT_POLL**.

int exs_evt_errno Set to indicate the status of the operation. An error indicated by a non-zero value also implicitly means that this socket has been deregistered from the event queue for all conditions.

exs_ahandle_t exs_ahandle
Set to the *ahandle* value passed at registration time.

int exs_socket Set to the socket referenced by the *exs_fildes* parameter passed at registration time.

union exs_evt_u exs_evt_union
The *exs_evt_u* union contains the **exs_evt_poll** structure that contains at least the following member:

int exs_evt_events
Set to one or more of the conditions being monitored.

Some values that the *exs_evt_errno* member can be set to are:

0	A poll event has occurred.
[EBADF]	The value of *exs_evt_fildes* is no longer an open file descriptor.
[ENOTSOCK]	The value of *exs_evt_fildes* is no longer a socket file descriptor.

Events indicating one or more conditions for a socket will be delivered to the specified event queue if an event is "armed" for that socket and some specific activity occurs that "triggers" the event. An event is automatically disarmed by the implementation when it is triggered. The application can only influence how and when an event will trigger through the use of arming. The implementation is responsible for determining when a particular event will trigger.

Which activities will arm an event depends on the type of monitored conditions and the state of the socket as described below:

EXS_POLLIN notification will be armed on a socket:

- Whenever an application registers for this condition
- Whenever an application drains all the data available to be read from the socket; i.e., a call to *recv*() or similar returns less data than requested or fails with [EAGAIN]
- Whenever an application drains all the pending inbound connections on the listening socket; i.e., a call to *accept*() returns the last pending connection

EXS_POLLOUT notifications will be armed on a socket:

- Whenever an application registers for this condition
- Whenever an application fills the implementation-managed transmit buffer; i.e., a call to *send*() or similar fails with [EAGAIN]

Which activities will trigger an event depends on the type of monitored conditions and the state of the socket as described below.

EXS_POLLIN notification will trigger on a socket:

- Whenever an application registers for this condition and there is data to be read on the socket or, in the case of a listening socket, there is at least one pending inbound connection
- Whenever **EXS_POLLIN** notification is armed on the socket and data arrives that is not entirely consumed by pending *exs_recv*() calls or similar
- Whenever **EXS_POLLIN** notification is armed on the listening socket and an inbound connection arrives with no pending *exs_accept*() calls

EXS_POLLOUT notification will trigger on a socket:

- Whenever an application registers for this condition and the implementation-managed transmit buffer is not full
- Whenever **EXS_POLLOUT** notification is armed on the socket and the transmit buffer becomes writable again

The **EXS_POLLIN** event for inbound connection indication is added for compatibility with the existing *poll*() implementation. It is not anticipated that an application will mix the use of *exs_accept*() and **EXS_POLLIN** on the same socket, although the text above indicates how this should function. In essence, pending *exs_accept*() calls will collect new connections in preference to the delivery of **EXS_POLLIN** notification.

Applications are expected to completely drain or fill the implementation-managed socket buffer when they receive either an **EXS_POLLIN** or an **EXS_POLLOUT** notification event for a socket used for data transfer. Failure to do so will prevent further notification events from being delivered without a new registration from the application.

Applications may mix the use of *exs_recv*() calls and **EXS_POLLIN** or *exs_send*() calls and **EXS_POLLOUT**. In both of these cases, the asynchronous calls will act as though an extra pool of socket buffer space or data for transmission is available to the implementation. This extra space or data will be used in advance of the normal socket buffer and, as a consequence, will affect the arming and triggering of **EXS_POLLIN** and **EXS_POLLOUT** events.

For example, the **EXS_POLLIN** event will be armed on a socket after an application registers for this condition and this socket has an empty receive socket buffer. The subsequent arrival of data on that socket would then trigger the **EXS_POLLIN** event. However, if the application calls *exs_recv*() before data arrives on the socket, the buffer space provided by the application in the *exs_recv*() call will be consumed before the normal socket buffer, and no **EXS_POLLIN** event will be delivered if the arriving data fits within the pre-posted buffer.

RETURN VALUE

The *exs_poll*() function returns the number of **exs_pollfd_t** entries successfully processed in the *fds* array. If the value returned is different from *nfds*, *errno* will be set to indicate the error.

ERRORS

[EAFNOSUPPORT]	The descriptor field in an **exs_pollfd_t** structure is not a supported socket family.
[EBADF]	The descriptor field in an **exs_pollfd_t** structure is not an open file descriptor.
[EFAULT]	The address range of the array pointed to by *fds* is invalid.
[EINVAL]	The *qhandle* argument is not a valid event queue handle. The *nfds* argument is greater than {OPEN_MAX}.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOTSOCK]	The descriptor field in an **exs_pollfd_t** structure is not a socket.
[ENOTSUP]	The *flags* value is not supported.
[EOPNOTSUPP]	The descriptor field in an **exs_pollfd_t** structure is not a supported socket type.
[EPERM]	The *exs_init*() function must be called first.

EXAMPLES

The following example shows how the *exs_poll*() function could be used to write the main event processing loop to guarantee continued delivery of **EXS_POLLIN** and **EXS_POLLOUT** events without the need to constantly call *exs_poll*() to register for these same conditions again.

```
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/exs.h>

/* Useful variables for this example. */
exs_qhandle_t    myqhandle;
exs_event_t      myevent[2];
int              myeventcnt;
int              myloop;
char             myrxbuf[1024];
int              myrxfd;
char             mytxbuf[1024];
int              mytxfd;
exs_pollfd_t     mypollfd[2];
nfds_t           mynfdscnt;
int              myfd;
ssize_t          myxfercnt;

/* Sockets and event queue creation code not shown. */

/* Set socket descriptions in non-blocking mode. */
if ((fcntl(myrxfd, F_SETFL, fcntl(myrxfd, F_GETFL) | O_NONBLOCK) == -1)
|| (fcntl(mytxfd, F_SETFL, fcntl(mytxfd, F_GETFL) | O_NONBLOCK) == 01))
{
/* Error processing. */
}
/* Register condition set for both sockets. */
mypollfd[0].exs_fildes  = myrxfd;
mypollfd[0].exs_events  = EXS_POLLIN;
mypollfd[0].exs_ahandle = 0;
mypollfd[1].exs_fildes  = mytxfd;
mypollfd[1].exs_events  = EXS_POLLOUT;
mypollfd[1].exs_ahandle = 0;
mynfdscnt               = 2;

if (exs_poll(mypollfd, mynfdscnt, 0, myqhandle) != mynfdscnt)
{
/* Error processing: look at errno value. */
}

    . . .

/* Main event dequeueing loop. */
while ((myeventcnt = exs_qdequeue(myqhandle, myevent, 2, NULL)) > 0)
{

    /* Look at each dequeued event. */
    for (myloop=0; myloop < myeventcnt; myloop++)
    {

        switch(myevent[myloop].exs_evt_type)
        {

            case EXS_EVT_POLL:
```

```
                        /* Check if event error. */
                        if (myevent[myloop].exs_evt_errno != 0)
                        {
                        /* Error processing. */
                        }

                        myfd = myevent[myloop].exs_evt_socket;

                        /* Drain socket rx buffer for this condition. */
                        if (myevent[myloop].exs_evt_union.exs_evt_poll.exs_evt_events &
                            EXS_POLLIN)
                        {
                            while (recv(myfd, myrxbuf, sizeof(myrxbuf), 0) != -1)
                            {
                            /* Process data. */
                            }
                            /* Make sure we stopped because the buffer was empty. */
                            if (errno != EAGAIN)
                            {
                            /* Error in processing. */
                            }
                        }

                        /* Fill up socket tx buffer for this condition. */
                        if (myevent[myloop].exs_evt_union.exs_evt_poll.exs_evt_events &
                             EXS_POLLOUT)
                        {
                            do
                            {
                            /* Prepare data. */
                            } while (send(myfd, mytxbuf, sizeof(mytxbuf), 0) != 1);

                            /* Make sure we stopped because the buffer was full. */
                            if (errno != EAGAIN)
                            {
                            /* Error processing. */
                            }
                        }
                        break;

                    default:
                    /* Error processing. */
                };
            }
        }
```

SEE ALSO

exs_accept(), *exs_init*(), *exs_qcreate*(), *exs_qdequeue*(), *exs_recv*(), *exs_send*()

exs_qcreate()

NAME

exs_qcreate – create an event queue

SYNOPSIS

```
#include <sys/exs.h>

exs_qhandle_t exs_qcreate (int depth)
```

DESCRIPTION

The *exs_qcreate*() function takes the following argument:

depth Specifies the guaranteed minimum number of events that can be stored in the queue. A value of 0 lets the implementation select a default minimum depth.

The *exs_qcreate*() function creates an event queue. Event queues are abstract objects used to store events. The **exs_qhandle_t** value returned by *exs_qcreate*() is the identifier or "handle" of the newly created queue. The handle is selected by the implementation and its value should not be interpreted or modified by the application. The handle must be used for all subsequent operations on that queue such as:

- Setting and retrieving various queue attributes with *exs_qmodify*() and *exs_qstatus*() respectively
- Dequeuing stored events with *exs_qdequeue*()
- Closing the queue with *exs_qdelete*()

The newly created queue is guaranteed to be able to store at least *depth* number of events. No events will be lost because the event queue is full; instead, an error will be returned at the time a call is made that would generate an event if a potential event queue overflow were possible.

The *exs_qcreate*() function can be called multiple times by a process with the effect of creating multiple event queues for this process. Event queue handles are valid only in the context of the process that called *exs_qcreate*(), and not in any child process created by *fork*() or after *exec*().

RETURN VALUE

If the operation is successful, *exs_qcreate*() returns the handle of the newly created event queue. Otherwise, *exs_qcreate*() returns **EXS_QHANDLE_INVALID**, and *errno* is set to indicate the error.

ERRORS

[EAGAIN]	The allocation of internal resources failed but a subsequent request may succeed.
[EINVAL]	Event queue resources were exceeded (e.g., the requested *depth* might be too large, …).
[EPERM]	The *exs_init*() function must be called first.

SEE ALSO

exs_init(), *exs_qdelete*(), *exs_qdequeue*(), *exs_qmodify*(), *exs_qstatus*()

exs_qdelete()

NAME

exs_qdelete – delete an event queue

SYNOPSIS

```
#include <sys/exs.h>

int exs_qdelete (exs_qhandle_t qhandle)
```

DESCRIPTION

The *exs_qdelete*() function takes the following argument:

qhandle Specifies an event queue handle previously obtained with a call to *exs_qcreate()*.

The *exs_qdelete*() function removes the event queue specified by *qhandle* and all resources associated with it.

Any events currently on the event queue will be lost if the *exs_qdelete*() function returns successfully.

The *exs_qdelete*() function will fail if there are any pending operations that cannot be internally cancelled. Therefore, the user should ensure that all operations have completed before calling the *exs_qdelete*() function. This may require waiting until all the completion events have been retrieved from the event queue with calls to the *exs_qdequeue()* function. It must be noted that the pending operations might be forced to complete sooner by calling the *exs_cancel()* function or closing all the file descriptors associated with these pending operations.

RETURN VALUE

If the operation is initiated successfully, *exs_qdelete()* returns 0. Otherwise, *exs_qdelete*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EBUSY]	There are still asynchronous operations in progress for this event queue.
[EINVAL]	The *qhandle* is not a valid event queue.
[EPERM]	The *exs_init()* function must be called first.

SEE ALSO

exs_cancel(), *exs_init()*, *exs_qcreate()*, *exs_qdequeue()*

exs_qdequeue()

NAME

exs_qdequeue – retrieve events from an event queue

SYNOPSIS

```
#include <sys/exs.h>

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

DESCRIPTION

The *exs_qdequeue*() function takes the following arguments:

qhandle Specifies an event queue handle previously obtained with a call to *exs_qcreate*().

evtvec Specifies the address of an array of **exs_event_t** types.

evtvec_cnt Specifies the number of **exs_event_t** elements in the *evtvec* array.

timeout Specifies how long the call should wait before timing out.

The **exs_event_t** type contains at least the following members:

```
int              exs_evt_type
int              exs_evt_errno    /* Status: 0 means success */
exs_ahandle_t    exs_evt_ahandle  /* Set to value passed by */
                                  /* application */
int              exs_evt_socket
union exs_evt_u  exs_evt_union
```

The *exs_evt_u* union contains at least the following members:

```
exs_evt_poll_t      exs_evt_poll
exs_evt_accept_t    exs_evt_accept
exs_evt_xfer_t      exs_evt_xfer
exs_evt_xfermsg_t   exs_evt_xfermsg
exs_evt_sendfile_t  exs_evt_sendfile
```

The *exs_qdequeue*() function is used to retrieve events from an event queue. The number of events actually dequeued and stored in *evtvec* is specified by the return value of the call and can be less than the supplied parameter *evtvec_cnt*.

When the call is successful, the type of each event stored in *evtvec* can be determined by looking at the value of the member *exs_evt_type* of the **exs_event_t** type. The interpretation of the rest of the content of each event depends on the event type and is described in the individual reference pages of the EXS calls that trigger these events.

If the *timeout* parameter is a NULL pointer, the call will block indefinitely until at least one event is available to be dequeued (blocking mode). If the *timeout* parameter is a non-NULL pointer and the members *tv_sec* and *tv_usec* of the **timeval** structure are set to 0 seconds and 0

milliseconds, the call returns immediately whether events were queued or not (polling mode). Otherwise, if the time limit expires before an event is available to be dequeued, a value of 0 is returned (timer mode).

Multiple threads may call the *exs_qdequeue*() function for the same event queue at the same time. When this happens, there is no requirement that the thread that initiated the operation be the same thread that is notified of the result. Which threads are given which results is implementation-dependent; however, each event notification will be delivered to only one thread.

RETURN VALUE

If the operation is successful, *exs_qdequeue*() returns number of events dequeued. Otherwise, *exs_qdequeue*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EFAULT]	The address range specified by *evtvec* and *evtvec_cnt* is not valid.
[EFAULT]	The address *timeout* is not valid.
[EINTR]	The call was interrupted.
[EINVAL]	The *qhandle* is not a valid event queue.
[EINVAL]	The *evtvec_cnt* value is less than 1 or greater that **EXS_EVTVEC_MAX**.
[EPERM]	The *exs_init*() function must be called first.

SEE ALSO

exs_init(), *exs_qcreate*()

exs_qmodify()

NAME

exs_qmodify – modify event queue attributes

SYNOPSIS

```
#include <sys/exs.h>

int exs_qmodify (exs_qhandle_t  qhandle,
                 int            attr_type,
                 void          *attr_value,
                 size_t         attr_length)
```

DESCRIPTION

The *exs_qmodify*() function takes the following arguments:

qhandle Specifies an event queue handle previously obtained with a call to *exs_qcreate*().

attr_type Specifies the attribute to modify and must be at least one of the following: **EXS_QATTR_DEPTH** or **EXS_QATTR_SIGNAL**.

attr_length Specifies the length of the attribute to modify.

attr_value Specifies the location where the value of the attribute to modify is stored.

The *exs_qmodify*() function is used to modify one attribute of an event queue. The modifiable queue attributes are the queue depth and the queue signal.

The queue depth is the guaranteed minimum number of events that can be stored in the queue. It is set initially when calling *exs_qcreate*() and can be retrieved by calling *exs_qstatus*(). Its value is modified by calling *exs_qmodify*() with *attr_type* set to **EXS_QATTR_DEPTH**, *attr_length* set to *sizeof*(*int*), and *attr_value* pointing to an object of type **int** containing the requested queue depth.

No events will be lost as a result of modifying the queue depth to a value lower than the current number of pending events; i.e., events already stored in the queue or associated with operations that have not yet completed.

When the *attr_type* is **EXS_QATTR_SIGNAL**, *attr_value* points to an **exs_signal_t** type, which specifies signal information associated with the queue. The *attr_length* parameter must be the size of an **exs_signal_t** type.

The **exs_signal_t** type contains at least the following members:

```
int exs_sigstate
int exs_signo
```

The *exs_sigstate* member can be set to **EXS_SIG_ENABLE** or **EXS_SIG_DISABLE**. Setting *exs_sigstate* to **EXS_SIG_DISABLE** disables queue signal generation. When *exs_sigstate* is set to **EXS_SIG_ENABLE**, queue signal generation is enabled. In that case, *exs_signo* must be set to the signal that will be generated for the process when certain queue conditions arise, as explained below.

Queue signal generation is initially disabled when the queue is created. When queue signal generation is enabled, the specified signal will be generated whenever the event queue is empty and an event is enqueued. A signal is also generated whenever *exs_sigstate* is set to **EXS_SIG_ENABLE** and there are already events pending on the queue.

RETURN VALUE

If the operation is successful, *exs_qmodify*() returns 0. Otherwise, *exs_qmodify*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EAGAIN]	The allocation of internal resources failed but a subsequent request may succeed.
[EFAULT]	The address range specified by *attr_value* and *attr_length* is not valid.
[EINVAL]	The *qhandle* is not a valid event queue.
[EINVAL]	The *attr_type* is not a valid event queue attribute.
[EINVAL]	The *attr_length* is not the exact size for the attribute *attr_type*.
[EINVAL]	The type **exs_signal_t** pointed to by *attr_value* has an incorrect content.
[EINVAL]	Event queue resources were exceeded (e.g., the requested queue depth might be too large, …).
[EPERM]	The *exs_init*() function must be called first.

SEE ALSO

exs_init(), *exs_qcreate*(), *exs_qstatus*()

exs_qstatus()

NAME

exs_qstatus – retrieve event queue attributes

SYNOPSIS

```
#include <sys/exs.h>

int exs_qstatus(exs_qhandle_t  qhandle,
                int            attr_type,
                void          *attr_value,
                size_t         attr_length)
```

DESCRIPTION

The *exs_qstatus*() function takes the following arguments:

qhandle Specifies an event queue handle previously obtained with a call to *exs_qcreate*().

attr_type Specifies the attribute to retrieve and must be at least one of the following: **EXS_QATTR_DEPTH**, **EXS_QATTR_SIGNAL,** or **EXS_QATTR_EVENTS**.

attr_length Specifies the length of the attribute to retrieve.

attr_value Specifies the location where the retrieved attribute should be written.

The *exs_qstatus*() function is used to retrieve one attribute of an event queue. The retrievable queue attributes are the queue depth, the queue signal, and the number of events currently stored in the queue.

The queue depth is the guaranteed minimum number of events that can be stored in the queue. It is set initially when calling *exs_qcreate*() and can be modified by calling *exs_qmodify*(). Its value is retrieved by calling *exs_qstatus*() with *attr_type* set to **EXS_QATTR_DEPTH**, *attr_length* set to *sizeof*(*int*), and *attr_value* set to a valid address where the queue depth will be returned.

If *attr_type* is **EXS_QATTR_SIGNAL**, the queue signal information is returned in an **exs_signal_t** type. The *attr_value* parameter must point to an **exs_signal_t** type, and *attr_length* must be the size of an **exs_signal_t** type. See the <exs.h> reference page for a description of the **exs_signal_t** type.

The queue events is the number of events currently stored in the queue. It is retrieved by calling *exs_qstatus*() with *attr_type* set to **EXS_QATTR_EVENTS**, *attr_length* set to *sizeof*(*int*), and *attr_value* set to a valid address where the number of events will be returned.

RETURN VALUE

Upon successful completion, a value of 0 is returned. On error, a value of -1 is returned and *errno* will be set to indicate the error.

ERRORS

[EFAULT] The address range specified by *attr_value* and *attr_length* is not valid.

[EINVAL]	The *qhandle* is not a valid event queue.
[EINVAL]	The *attr_type* is not a valid event queue attribute.
[EINVAL]	The *attr_length* is not the exact size for the attribute *attr_type*.
[EPERM]	The *exs_init*() function must be called first.

SEE ALSO

exs_init(), *exs_qcreate*(), *exs_qmodify*()

exs_recv()

NAME

exs_recv – asynchronously receive a message from a connected socket

SYNOPSIS

```
#include <sys/exs.h>

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)
```

DESCRIPTION

The *exs_recv*() function will receive a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

The *exs_recv*() function takes the following arguments:

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 the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG_PEEK Peeks at the incoming message. The data is treated as unread and the next *exs_recv*() or similar function will still return this data.

MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL On **SOCK_STREAM** sockets this requests that the function wait until the full amount of incoming data has been copied to the supplied buffer before a completion event is posted to the *qhandle* event queue. The function may return a smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if **MSG_PEEK** was specified, or if an error is pending for the socket.

qhandle Specifies the destination event queue on which the completion event should be posted. This event queue must have previously been created using *exs_qcreate*().

ahandle Specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request. The application may use this value to identify this request after retrieving the completion event by calling *exs_qdequeue*() or when calling *exs_cancel*().

mhandle Specifies a registered memory handle, if any, associated with the *buffer*. If *mhandle* is different than **EXS_MHANDLE_UNREGISTERED**, it must be a valid handle previously obtained by calling *exs_mregister*() for a region of memory that encompasses at least the buffer where the incoming data is to be stored.

The length of the buffer that is supplied for the incoming message is specified by the *length* argument.

For message-based sockets, such as **SOCK_DGRAM** and **SOCK_SEQPACKET**, one entire message will be read in a single operation. If a message is too long to fit in the supplied buffer, and **MSG_PEEK** is not set in the *flags* argument, the excess bytes will be discarded.

For stream-based sockets, such as **SOCK_STREAM**, message boundaries will be ignored. In this case, data will be returned to the user as soon as it becomes available, and no data will be discarded.

Once the incoming message has been copied into the supplied buffer, a completion event will be placed on the event queue that was specified by the *qhandle* argument. The completion event can be retrieved from the event queue with the *exs_qdequeue*() function. The completion event for the *exs_recv*() operation is the **exs_event** structure. This structure is defined in the **<sys/exs.h>** header file and it contains at least the following members:

int exs_evt_type Set to **EXS_EVT_RECV**.

int exs_evt_errno Set to indicate the status of the I/O operation.

exs_ahandle_t exs_evt_ahandle
Set to the value of the *ahandle* argument.

int exs_evt_socket Set to the value of the *socket* argument.

union exs_evt_u exs_evt_union
The *exs_evt_u* union contains the **exs_evt_xfer** structure that contains at least the following members:

*void *exs_evt_buffer*
Set to the value of the *buffer* argument.

size_t exs_evt_length
Set to the number of bytes received.

exs_mhandle_t exs_evt_mhandle
Set to the value of the *mhandle* argument.

Possible values that could be returned in the *exs_evt_errno* field include:

0 The asynchronous receive operation was successful.

[ECANCELED] The operation has been cancelled.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] The system detected an address that was not valid while attempting to access the *buffer* argument.

If the socket referenced by the *socket* argument is closed before the receive operation is completed, an event will be delivered to the event queue specified by the *qhandle* argument indicating the failure.

The application should not access the memory buffer that is described by *buffer* argument until the completion event is retrieved by calling *exs_qdequeue*().

Calls to the *exs_recv*() function can be interleaved with other synchronous (*recv*(), *recvmsg*(), *read*(), etc.) and asynchronous (e.g., *exs_recvmsg*()) receive calls but this is not recommended. The data will be delivered to the application in the order supplied by the underlying transport, but the ordering of outstanding asynchronous and synchronous receive operations is undefined.

The socket option **SO_ASYNC_RECV_ORDERED** controls whether asynchronous receive data buffers will be returned in the order posted by the application. The default value will be **FALSE**.

The *exs_recv*() function will not be affected by the **O_NONBLOCK** flag.

RETURN VALUE

The *exs_recv*() will return one of the following values:

0 The *exs_recv*() operation has been initiated. A completion event will be posted to the event queue that was specified by the *qhandle* argument.

-1 The *exs_recv*() operation has failed, and *errno* is set to indicate the error.

ERRORS

The *exs_recv*() function **will** fail if:

[EBADF] The *socket* argument is not a valid file descriptor.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] The system detected an address that was not valid while attempting to access the *buffer* argument.

[EINVAL] The *qhandle* is not a valid event queue, the *mhandle* is not a valid memory registration handle, or the **MSG_OOB** flag is set and no out-of-band data is available.

[ENOTCONN] A receive is attempted on a connection-mode socket that is not connected.

[ENOTSOCK] The *socket* argument does not refer to a socket.

[EOPNOTSUPP] The specified flags are not supported for this socket type or protocol.

[EPERM] The *exs_init*() function must be called first.

[ETIMEDOUT] The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *exs_recv*() function **may** fail if:

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

RATIONALE

The return value of *exs_recv* is type **ssize_t** in order to permit implementations to support a synchronous operation as well as asynchronous behavior.

If a synchronous behavior is supported and the synchronous operation is successful, *exs_recv*() returns the number of bytes received. Otherwise, *exs_recv*() returns -1, and *errno* is set to indicate the error. This feature is available only in implementations that document it, and only if the option is selected by the application by setting an implementation-specified value in the *flags* argument.

SEE ALSO

exs_cancel(), *exs_init*(), *exs_mregister*(), *exs_qcreate*(), *exs_qdequeue*(), *exs_recvmsg*()

exs_recvmsg()

NAME

exs_recvmsg – asynchronously receive a message from a socket

SYNOPSIS

```
#include <sys/exs.h>

ssize_t exs_recvmsg(int             socket,
                    exs_msghdr_t   *message,
                    int             flags,
                    exs_qhandle_t   qhandle,
                    exs_ahandle_t   ahandle)
```

DESCRIPTION

The *exs_recvmsg*() function will receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless sockets because it permits the application to retrieve the source address of received data.

The *exs_recvmsg*() function takes the following arguments:

socket Specifies the socket file descriptor.

message Points to an **exs_msghdr** structure, containing both the buffer to store the source address and the buffers for the incoming message.

flags Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG_PEEK Peeks at the incoming message. The data is treated as unread and the next *exs_recv*() or similar function will still return this data.

MSG_OOB Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.

MSG_WAITALL On **SOCK_STREAM** sockets this requests that the function wait until the full amount of incoming data has been copied to the supplied buffers before a completion event is posted to the *qhandle* event queue. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if **MSG_PEEK** was specified, or if an error is pending for the socket.

qhandle Specifies the destination event queue on which the completion event should be posted. This event queue must have previously been created using *exs_qcreate*().

ahandle Specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request. The application may use this

value to identify this request after retrieving the completion event by calling *exs_qdequeue*() or when calling *exs_cancel*().

The **exs_msghdr** structure is defined in the **<sys/exs.h>** header file and contains at least the following members:

```
void          *msg_name
socklen_t      msg_namelen
exs_iovec_t   *msg_iov
int            msg_iovlen
void          *msg_control
socklen_t      msg_controllen
int            msg_flags
```

The *msg_name* and *msg_namelen* fields of the *message* specify the source address if the socket is unconnected. The length and format of the address depend on the address family of the socket. If the socket is connected, the *msg_name* and *msg_namelen* fields are ignored. The *msg_name* field may be a null pointer if no names are desired or required.

The *msg_iov* and *msg_iovlen* fields are used to specify where the received data will be stored. The *msg_iov* field points to an array of *exs_iovec* structures; the *msg_iovlen* field will be set to the dimension of this array. The **exs_iovec** structure is defined in the **<sys/exs.h>** header file and contains at least the following members:

```
void           *iov_base
size_t          iov_len
exs_mhandle_t   iov_mhandle
```

In each **exs_iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn until all of the received data is stored or all of the areas have been filled. The *iov_mhandle* field specifies a registered memory handle, if any, associated with the storage area specified by the *iov_base* field. If *iov_mhandle* is different than **EXS_MHANDLE_UNREGISTERED**, it must be a valid handle previously obtained by calling *exs_mregister*() for a region of memory that encompasses at least the storage area containing the message to receive.

The *msg_control* and *msg_controllen* fields may be used for protocol-dependent ancillary data in the same fashion as the *msg_control* and *msg_controllen* fields of the **msghdr** structure in a *recvmsg*() call.

The *msg_flags* field in the *message* is ignored on input, but may contain meaningful values on output. Upon successful completion, the *msg_flags* field will be the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:

MSG_EOR End-of-record was received (if supported by the protocol).

MSG_OOB Out-of-band data was received.

MSG_TRUNC Normal data was truncated.

MSG_CTRUNC Control data was truncated.

For message-based sockets – such as **SOCK_DGRAM** and **SOCK_SEQPACKET** – one entire message will be read in a single operation. If a message is too long to fit in the supplied buffers,

and **MSG_PEEK** is not set in the *flags* argument, the excess bytes will be discarded, and **MSG_TRUNC** will be set in the *msg_flags* field of the **exs_msghdr** structure.

For stream-based sockets – such as **SOCK_STREAM** – message boundaries will be ignored. In this case, data will be returned to the user as soon as it becomes available, and no data will be discarded.

Once the receive operation has been completed, a completion event will be placed on the event queue that was specified by the *qhandle* argument. The completion event can be retrieved from the event queue with the *exs_qdequeue()* function. The completion event for the *exs_recvmsg*() operation is the **exs_event** structure. This structure is defined in the **<sys/exs.h>** header file and contains at least the following members:

int exs_evt_type Set to **EXS_EVT_RECVMSG**.

int exs_evt_errno Set to indicate the status of the I/O operation.

exs_ahandle_t exs_evt_ahandle
Set to the value of the *ahandle* argument.

int exs_evt_socket Set to the value of the *socket* argument.

union exs_evt_u exs_evt_union
The *exs_evt_u* union contains the **exs_evt_xfermsg** structure that contains at least the following members:

*exs_msghdr_t *exs_evt_msg*
Set to the value of the *message* argument.

size_t exs_evt_length
Set to the number of bytes received.

Possible values that could be returned in the *exs_evt_errno* field include:

0 The asynchronous receive operation was successful.

[ECANCELED] The operation has been cancelled.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] The system detected an address that was not valid while attempting to access the *message* argument, or the *msg_name*, the *msg_iov*, or the *msg_control* fields in the **exs_msghdr** structure or the *iov_base* field in the **exs_iovec** structure.

If the socket referenced by the *socket* argument is closed before the receive operation is completed, an event will be delivered to the event queue specified by the *qhandle* argument indicating the failure.

The application should not access any of the memory buffers that are described by the *message* argument until the completion event is retrieved by calling *exs_qdequeue()*.

Calls to the *exs_recvmsg*() function can be interleaved with other synchronous (*recv*(), *recvmsg*(), *read*(), etc.) and asynchronous (e.g., *exs_recv()*) receive calls, but this is not

recommended. The data will be delivered to the application in the order supplied by the underlying transport, but the ordering of outstanding asynchronous and synchronous receive operations is undefined.

The socket option **SO_ASYNC_RECV_ORDERED** controls whether asynchronous receive data buffers will be returned in the order posted by the application. The default value will be FALSE. The socket option **SO_RCVLOWAT** affects *exs_recvmsg*() as specified in *setsockopt*().

The *exs_recvmsg*() function will not be affected by the **O_NONBLOCK** flag.

RETURN VALUE

The *exs_recvmsg*() function will return one of the following values:

0	The *exs_recvmsg*() operation has been initiated. A completion event will be posted to the event queue that was specified by the *qhandle* argument.
-1	The *exs_recvmsg*() operation has failed, and *errno* is set to indicate the error.

ERRORS

The *exs_recvmsg*() function **will** fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EFAULT]	The system detected an address that was not valid while attempting to access the *message* argument, the *msg_name*, the *msg_iov*, or the *msg_control* fields in the **exs_msghdr** structure, or the *iov_base* field in the **exs_iovec** structure.
[EINVAL]	The *qhandle* is not a valid event queue, the *iov_mhandle* is not a valid memory registration handle, the sum of the *iov_len* values overflows a **ssize_t**, or the **MSG_OOB** flag is set and no out-of-band data is available.
[EMSGSIZE]	The *msg_iovlen* field of the **exs_msghdr** structure pointed to by the *message* argument is less than or equal to 0, or is greater than {**IOV_MAX**}.
[ENOTCONN]	A receive is attempted on a connection-mode socket that is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The specified flags are not supported for this socket type or protocol.
[EPERM]	The *exs_init*() function must be called first.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *exs_recvmsg*() function **may** fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

RATIONALE

The return value of *exs_recvmsg* is type **ssize_t** in order to permit implementations to support a synchronous operation as well as asynchronous behavior.

If a synchronous behavior is supported and the synchronous operation is successful, *exs_recvmsg*() returns the number of bytes received. Otherwise, *exs_recvmsg*() returns -1, and *errno* is set to indicate the error. This feature is available only in implementations that document it, and only if the option is selected by the application by setting an implementation-specified value in the *flags* argument.

SEE ALSO

exs_cancel(), *exs_init*(), *exs_mregister*(), *exs_qcreate*(), *exs_qdequeue*(), *exs_recv*()

exs_send()

NAME

exs_send – asynchronously send a message on a socket

SYNOPSIS

```
#include <sys/exs.h>

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

DESCRIPTION

The *exs_send*() function initiates an asynchronous transmission of a message from the specified socket to its peer. The *exs_send*() function will send a message only when the socket is connected (including when the peer of a connectionless socket has been set via *connect*() or *exs_connect*()).

The *exs_send*() function takes the following arguments:

socket Specifies the socket file descriptor.

buffer Points to a buffer containing the message to send.

length Specifies the length of the message in bytes.

flags Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG_EOR Terminates a record (if supported by the protocol).

MSG_OOB Sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.

qhandle Specifies the destination event queue on which the completion event should be posted. This event queue must have previously been created using *exs_qcreate*().

ahandle Specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request. The application may use this value to identify this request after retrieving the completion event by calling *exs_qdequeue*() or when calling *exs_cancel*().

mhandle Specifies a registered memory handle, if any, associated with the *buffer*. If *mhandle* is different than **EXS_MHANDLE_UNREGISTERED**, it must be a valid handle previously obtained by calling *exs_mregister*() for a region of memory that encompasses at least the buffer containing the message to send.

The length of the message to be sent is specified by the *length* argument. If the message is too long to pass through the underlying protocol, *exs_send*() will fail and no data will be transmitted.

Successful completion of a call to *exs_send*() does not guarantee delivery of the message. The function will return when transmission has been initiated or, at a minimum, when the request has been queued. A return value of -1 indicates only locally-detected errors.

Once the send operation has been completed, a completion event will be placed on the event queue that was specified by the *qhandle* argument. The completion event can be retrieved from the event queue with the *exs_qdequeue*() function. The completion event for the *exs_send*() operation is the **exs_event** structure. This structure is defined in the **<sys/exs.h>** header file and it contains at least the following members:

int exs_evt_type Set to **EXS_EVT_SEND**.

int exs_evt_errno Set to indicate the status of the I/O operation.

exs_ahandle_t exs_evt_ahandle
Set to the value of the *ahandle* argument.

int exs_evt_socket Set to the value of the *socket* argument.

union exs_evt_u exs_evt_union
The *exs_evt_u* union contains the **exs_evt_xfer** structure that contains at least the following members:

*void *exs_evt_buffer*
Set to the value of the *buffer* argument.

size_t exs_evt_length
Set to the number of bytes transmitted.

exs_mhandle_t exs_evt_mhandle
Set to the value of the *mhandle* argument.

Possible values that could be returned in the *exs_evt_errno* field include:

0 The asynchronous send operation was successful.

[ECANCELED] The operation has been cancelled.

[ECONNRESET] A connection was forcibly closed by a peer.

[EFAULT] The system detected an address that was not valid while attempting to access the *buffer* argument.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type

SOCK_STREAM, the **SIGPIPE** signal is generated to the calling thread.

If the socket referenced by the *socket* argument is closed before the send operation is completed, an event will be delivered to the event queue specified by the *qhandle* argument indicating the failure.

The application should not access the memory buffer that is described by the *buffer* argument until the completion event is retrieved by calling *exs_dequeue()*.

Calls to the *exs_send*() function can be interleaved with other synchronous (*send*(), *sendmsg*(), *write*(), etc.) and asynchronous (e.g., *exs_sendmsg()*) transmit calls. For a given thread, the data will be delivered to the transport in the order posted to the API. In the case of multiple threads, the application must take the usual precautions to guarantee proper ordering.

The *exs_send*() function will not be affected by the **O_NONBLOCK** flag.

The socket in use may require the process to have appropriate privileges to use the *exs_send*() function.

RETURN VALUE

The *exs_send*() function will return one of the following values:

0	The *exs_send*() operation has been initiated. A completion event will be posted to the event queue that was specified by the *qhandle* argument.
-1	The *exs_send*() operation has failed, and *errno* is set to indicate the error.

ERRORS

The *exs_send*() function **will** fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EDESTADDRREQ]	The socket is not connection-mode and no peer address is set.
[EFAULT]	The system detected an address that was not valid while attempting to access the *buffer* argument.
[EINVAL]	The *qhandle* is not a valid event queue or the *mhandle* is not a valid memory registration handle.
[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
[ENOTCONN]	The specified socket is not connected or otherwise has not had the peer pre-specified.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
[EPERM]	The *exs_init()* function must be called first.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated to the calling thread.

The *exs_send*() function **may** fail if:

[EACESS] The calling process does not have the appropriate privileges.

[EIO] An I/O error occurred while reading from or writing to the file system.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

RATIONALE

The return value of *exs_send*() is type **ssize_t** in order to permit implementations to support a synchronous operation as well as asynchronous behavior.

If a synchronous behavior is supported and the synchronous operation is successful, *exs_send*() returns the number of bytes sent. Otherwise, *exs_send*() returns -1, and *errno* is set to indicate the error. This feature is available only in implementations that document it, and only if the option is selected by the application by setting an implementation-specified value in the *flags* argument.

SEE ALSO

exs_cancel(), *exs_connect*(), *exs_init*(), *exs_mregister*(), *exs_qcreate*(), *exs_qdequeue*(), *exs_sendmsg*()

exs_sendfile()

NAME

exs_sendfile – initiate transmission of the contents of a file from a socket

SYNOPSIS

```
#include <sys/exs.h>

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

DESCRIPTION

The *exs_sendfile*() function initiates the transmission of the contents of one or more files and memory buffers from a socket. The socket must be connected, including when the peer address of a connectionless socket has been set by calling *connect*() or *exs_connect*().

The *exs_sendfile*() function takes the following arguments:

socket Specifies the connected socket file descriptor.

sendvec Specifies an array of file descriptors and memory buffers.

sendvec_cnt Specifies the number of elements in the *sendvec* array.

flags Specifies options that apply to the operation.

qhandle Specifies the destination event queue on which the completion event should be posted. The event queue must have been created previously using *exs_qcreate*().

ahandle Specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request.

Each **exs_xferfile_t** structure in the *sendvec* array contains at least the following members:

```
int               exs_xfv_type
union exs_xfvec   exs_xfv
```

The *exs_xfv_type* member determines which member of the *exs_xfv* union is used. The *exs_xfvec_type* member can have at least any of the values **EXS_IOVEC** and **EXS_FDVEC**.

The *exs_xfvec* union contains at least the following members:

```
exs_iovec_t    exs_iov
exs_fdvec_t    exs_fdv
```

The **exs_iovec_t** structure contains at least the following members:

```
void          *iov_base
size_t         iov_len
exs_mhandle_t  iov_mhandle
```

The **exs_fdvec_t** structure contains at least the following members:

```
int         exs_fildes
off_t       exs_offset
size_t      exs_length
int         exs_flags
```

All values in the *sendvec* array should be valid when *exs_sendfile*() is called. However, implementations may defer checking for errors until it becomes necessary to initiate the transfer of each individual *sendvec* element.

The aggregate data specified by the *sendvec* array is transmitted in the order that it appears in the array.

For an **exs_iovec_t** structure, *exs_sendfile*() transfers *iov_len* bytes starting at the address specified by *iov_base*. *If iov_len*, *iov_base*, or the address range specified by *iov_base* and *iov_len* is invalid when the **exs_iovec_t** structure is initially checked for errors, the *exs_sendfile*() completion event will indicate an error. If the address range becomes invalid later, the results are implementation-specific.

For an **exs_fdvec_t** structure, *exs_sendfile*() transfers up to *exs_length* bytes or until the end-of-file is reached, starting at *exs_offset* bytes from the beginning of the file. If *exs_length* is zero, *exs_sendfile*() transfers data until the end-of-file is reached. If *exs_fildes* or *exs_offset* is invalid when the **exs_fdvec_t** structure is initially checked for errors, the *exs_sendfile*() completion event will indicate an error. If any of the values of the **exs_fdvec_t** structure becomes invalid later, the results are implementation-specific. If *exs_fildes* does not specify a regular file, the results of the operation and the meaning of *exs_offset* are implementation-specific. If the contents of the file specified by *exs_fildes* are modified before the transfer is completed, the results are implementation-specific.

The *flags* argument is formed by logically OR'ing zero or more of the following flags:

EXS_SHUT_WR

EXS_SHUT_WR shuts down the socket for writing after all data has been passed successfully to the transport layer for transmission, as if the application called *shutdown*() with the **SHUT_WR** option. For some transport protocols (for example, TCP), this permits the peer application to recognize the end-of-transmission, but it still permits the local application to read any data that might have been sent from the peer. If an error is detected during the send operation, the socket is not shut down for writing.

The *ahandle* argument may be set to any arbitrary value. The application can use that value to identify this request when calling *exs_cancel*() and when retrieving the completion event by calling *exs_qdequeue*().

If the *socket* argument is closed or disconnected before the data transfer operation is completed, an event will be delivered to the relevant event queue indicating the failure.

When the asynchronous operation is complete, an **EXS_EVT_SENDFILE** completion event is put into the event queue specified by *qhandle*. The completion event can be retrieved by calling *exs_qdequeue*(). The **EXS_EVT_SENDFILE** event is returned in the **exs_event_t** structure, which has at least the following members:

```
int             exs_evt_type
int             exs_evt_errno
```

```
exs_ahandle_t    exs_evt_ahandle
int              exs_evt_socket
union exs_evt_u  exs_evt_union
```

The *exs_evt_u* union contains the **exs_evt_xferfile** structure that contains at least the following members:

```
exs_xferfile_t   *exs_evt_sendvec
int               exs_evt_sendvec_cnt
size_t            exs_evt_length
```

The *exs_evt_type* member is set to the value **EXS_EVT_SENDFILE**. The *exs_evt_length* member is set to the cumulative number of data bytes that were passed to the transport layer for transmission before the operation completed successfully or unsuccessfully. The *exs_evt_socket, exs_evt_ahandle, exs_evt_sendvec,* and *exs_evt_sendvec_cnt* members are set to the corresponding *exs_sendfile*() arguments.

The *exs_evt_errno* member is set to zero if the operation completed successfully; otherwise, *exs_evt_errno* is set to an appropriate value. Successful completion of the *exs_sendfile*() call and receipt of a successful **EXS_EVT_SENDFILE** event does not guarantee delivery of the data to the remote peer. They only indicate that no local error was detected. Some values that the *exs_evt_errno* member can be set to are:

0	The asynchronous send operation completed successfully.
[EACCES]	The process does not have the appropriate privileges.
[ECANCELED]	The asynchronous operation was cancelled.
[EFAULT]	The address range specified by a pair of *iov_base* and *iov_len* members is not valid.
[EINVAL]	The value of an *iov_mhandle* member is not valid.
[EIO]	An I/O error occurred while reading from the file specified by an *exs_fildes* member.
[ENETDOWN]	The local network interface used to reach the destination is down.
[ENETUNREACH]	There is no route to the network.
[EPIPE]	The socket is shut down for writing, or the connection was terminated by the remote peer. In the latter case, if the socket is type SOCK_STREAM, the SIGPIPE signal is generated. Note that the default action for SIGPIPE is to terminate the process, unless the process specified a signal handler for SIGPIPE or chose to ignore or block the signal.

RETURN VALUE

If the operation is initiated successfully, *exs_sendfile*() returns 0. Otherwise, *exs_sendfile*() returns -1, and *errno* is set to indicate the error.

ERRORS

[EACCES]	The process does not have the appropriate privileges.
[EAFNOSUPPORT]	The socket address family is unsupported.
[EAGAIN]	Too many asynchronous operations are in progress.
[EBADF]	The *socket* argument is not an open file descriptor, or an *exs_fildes* member is not a valid file descriptor open for reading.
[EDESTADDRREQ]	The socket is not connection-mode, and no peer address is set.
[EFAULT]	The address range specified by the *sendvec* and *sendvec_cnt* arguments or by a pair of *iov_base* and *iov_len* members is not valid.
[EINVAL]	The *qhandle* argument or an *iov_mhandle* member is not valid.
[ENETDOWN]	The local network interface used to reach the destination is down.
[ENETUNREACH]	There is no route to the network.
[ENOBUFS]	Insufficient system resources are available to perform the operation.
[ENOTCONN]	The *socket* argument is not connected or otherwise has not had the peer pre-specified.
[ENOTSOCK]	The *socket* argument is not a socket file descriptor.
[EOPNOTSUPP]	One or more of the values set in the *flags* argument are not supported.
[EPERM]	The *exs_init*() function had not been called first.
[EPIPE]	The socket is shut down for writing, or the connection was terminated by the remote peer. In the latter case, if the socket is type **SOCK_STREAM**, the **SIGPIPE** signal is generated. Note that the default action for **SIGPIPE** is to terminate the process, unless the process specified a signal handler for **SIGPIPE** or chose to ignore or block the signal.

RATIONALE

The return value of *exs_sendfile*() is type **ssize_t** in order to permit implementations to support a synchronous operation as well as asynchronous behavior.

If a synchronous behavior is supported and the synchronous operation is successful, *exs_sendfile*() returns the number of bytes sent. Otherwise, *exs_sendfile*() returns -1, and *errno* is set to indicate the error. This feature is available only in implementations that document it, and only if the option is selected by the application by setting an implementation-specified value in the *flags* argument.

The **exs_xferfile_t** structure is not updated by the operation to reflect the progress of the transmission. Consequently, if the operation is interrupted and a value of *exs_length* is zero in an **exs_fdvec_t** structure, an application cannot use *exs_evt_length* in the **exs_evt_sendfile_t**

structure to determine how much data was passed to the transport layer for transmission before the operation terminated and, therefore, where in the **exs_xferfile_t** structure to restart the operation, if desirable. Although restartability might be a lofty goal, it is only practical if the implementation updated the **exs_fdvec_t** structure to reflect acknowledged transmitted data, not merely the amount of data passed to the transport layer for transmission. That is not done typically even by implementations of existing synchronous socket send operations. Therefore, the overhead and the propriety of updating the **exs_xferfile_t** structure in order to permit restartability is dubious.

SEE ALSO

exs_cancel(), *exs_connect*(), *exs_init*(), *exs_qcreate*(), *exs_qdequeue*()

exs_sendmsg()

NAME

exs_sendmsg – asynchronously send a message on a socket

SYNOPSIS

```
#include <sys/exs.h>

ssize_t exs_sendmsg(int                   socket,
                    const exs_msghdr_t  *message,
                    int                   flags,
                    exs_qhandle_t         qhandle,
                    exs_ahandle_t         ahandle)
```

DESCRIPTION

The *exs_sendmsg*() function initiates an asynchronous transmission of a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message will be sent to the address specified by *exs_msghdr*. If the socket is connection-mode, the destination address in *exs_msghdr* will be ignored.

The *exs_sendmsg*() function takes the following arguments:

socket Specifies the socket file descriptor.

message Points to an **exs_msghdr** structure, containing both the destination address and the buffers for the outgoing message.

flags Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG_EOR Terminates a record (if supported by the protocol).

MSG_OOB Sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.

qhandle Specifies the destination event queue on which the completion event should be posted. This event queue must have previously been created using *exs_qcreate*().

ahandle Specifies an asynchronous handle, which is an arbitrary application-specific value that can be used to identify this request. The application may use this value to identify this request after retrieving the completion event by calling *exs_qdequeue*() or when calling *exs_cancel*().

The **exs_msghdr** structure is defined in the **<sys/exs.h>** header file and it contains at least the following members:

```
void           *msg_name
socklen_t       msg_namelen
exs_iovec_t    *msg_iov
int             msg_iovlen
void           *msg_control
```

```
socklen_t       msg_controllen
int             msg_flags
```

The *msg_name* and *msg_namelen* fields of the *message* are used to specify the destination address if the socket is connectionless-mode. The length and format of the address depend on the address family of the socket. If the socket is connection-mode, the *msg_name* and *msg_namelen* fields are ignored.

The *msg_iov* and *msg_iovlen* fields specify zero or more buffers containing the data to be sent. *msg_iov* points to an array of **exs_iovec** structures; the *msg_iovlen* field will be set to the dimension of this array. The **exs_iovec** structure is defined in the **<sys/exs.h>** header file and contains at least the following members:

```
void           *iov_base
size_t          iov_len
exs_mhandle_t   iov_mhandle
```

In each **exs_iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated by *msg_iov* is sent in turn. The *iov_mhandle* field specifies a registered memory handle, if any, associated with the storage area specified by the *iov_base* field. If *iov_mhandle* is different than **EXS_MHANDLE_UNREGISTERED**, it must be a valid handle previously obtained by calling *exs_mregister*() for a region of memory that encompasses at least the storage area containing the message to send.

The *msg_control* and *msg_controllen* fields may be used for protocol-dependent ancillary data in the same fashion as the *msg_control* and *msg_controllen* fields of the **msghdr** structure in a *sendmsg*() call.

The *msg_flags* field in the *message* is currently ignored.

If the message is too long to pass through the underlying protocol, *exs_sendmsg*() will fail and no data will be transmitted.

Successful completion of a call to *exs_sendmsg*() does not guarantee delivery of the message. The function will return when transmission has been initiated or, at a minimum, when the request has been queued. A return value of -1 indicates only locally-detected errors.

Once the send operation has been completed, a completion event will be placed on the event queue that was specified by the *qhandle* argument. The completion event can be retrieved from the event queue with the *exs_qdequeue*() function. The completion event for the *exs_sendmsg*() operation is the **exs_event** structure. This structure is defined in the **<sys/exs.h>** header file and it contains at least the following members:

int exs_evt_type Set to **EXS_EVT_SENDMSG**.

int exs_evt_errno Set to indicate the status of the I/O operation.

exs_ahandle_t exs_evt_ahandle
Set to the value of the *ahandle* argument.

int exs_evt_socket Set to the value of the *socket* argument.

union exs_evt_u exs_evt_union

The *exs_evt_u* union contains the **exs_evt_xfermsg** structure that contains at least the following members:

*exs_msghdr_t *exs_evt_msg*

Set to the value of the *message* argument.

size_t exs_evt_length

Set to the number of bytes transmitted.

Possible values that could be returned in the *exs_evt_errno* field include:

0	The asynchronous send operation was successful.
[ECANCELED]	The operation has been cancelled.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EFAULT]	The system detected an address that was not valid while attempting to access the *message* argument, or the *msg_name*, the *msg_iov*, or the *msg_control* fields in the **exs_msghdr** structure or the *iov_base* field in the **exs_iovec** structure.
[ENETDOWN]	The local network interface used to reach the destination is down.
[ENETUNREACH]	No route to the network is present.
[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated to the calling thread.

If the socket referenced by the *socket* argument is closed before the send operation is completed, an event will be delivered to the event queue specified by the *qhandle* argument indicating the failure.

The application should not access any of the memory buffers that are described by the *message* argument until the completion event is retrieved by calling *exs_qdequeue*().

Calls to the *exs_sendmsg*() function can be interleaved with other synchronous (*send*(), *sendmsg*(), *write*(), etc.) and asynchronous (e.g., *exs_send*()) transmit calls. For a given thread, the data will be delivered to the transport in the order posted to the API. In the case of multiple threads, the application must take the usual precautions to guarantee proper ordering.

The *exs_sendmsg*() function will not be affected by the **O_NONBLOCK** flag.

The socket in use may require the process to have appropriate privileges to use the *exs_sendmsg*() function.

RETURN VALUE

The *exs_sendmsg*() will return one of the following values:

0	The *exs_sendmsg*() operation has been initiated. A completion event will be posted to the event queue that was specified by the *qhandle* argument.

-1 The *exs_sendmsg*() operation has failed, and *errno* is set to indicate the error.

ERRORS

The *exs_sendmsg*() function **will** fail if:

[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EDESTADDRREQ]	The socket is not connection-mode and no peer address is set.
[EFAULT]	The system detected an address that was not valid while attempting to access the *message* argument, the *msg_name*, the *msg_iov*, or the *msg_control* fields in the **exs_msghdr** structure, or the *iov_base* field in the **exs_iovec** structure.
[EINVAL]	The *qhandle* is not a valid event queue, the *iov_mhandle* is not a valid memory registration handle, or the sum of the *iov_len* values overflows a **ssize_t**.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in *address*.
[EMSGSIZE]	The message is too large to be sent all at once (as the socket requires), or the *msg_iovlen* field of the **exs_msghdr** structure pointed to by the *message* argument is less than or equal to 0 or is greater than {**IOV_MAX**}.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters, or the pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
[ENOTCONN]	The socket is connection-mode but is not connected.
[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
[EPERM]	The *exs_init*() function must be called first.

[EPIPE] The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated to the calling thread.

The *exs_sendmsg*() function **may** fail if:

[EACESS] The calling process does not have the appropriate privileges.

[EDESTADDRREQ] The socket is not connection-mode and no peer address is set, and no destination address was specified.

[EHOSTUNREACH] The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EISCONN] A destination address was specified and the socket is already connected.

[ENETDOWN] The local network interface used to reach the destination is down.

[ENETUNREACH] No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

RATIONALE

The return value of *exs_sendmsg* is type **ssize_t** in order to permit implementations to support a synchronous operation as well as asynchronous behavior.

If a synchronous behavior is supported and the synchronous operation is successful, *exs_sendmsg*() returns the number of bytes sent. Otherwise, *exs_sendmsg*() returns -1, and *errno* is set to indicate the error. This feature is available only in implementations that document it, and only if the option is selected by the application by setting an implementation-specified value in the *flags* argument.

SEE ALSO

exs_cancel(), *exs_connect*(), *exs_init*(), *exs_mregister*(), *exs_qcreate*(), *exs_qdequeue*(), *exs_send*()