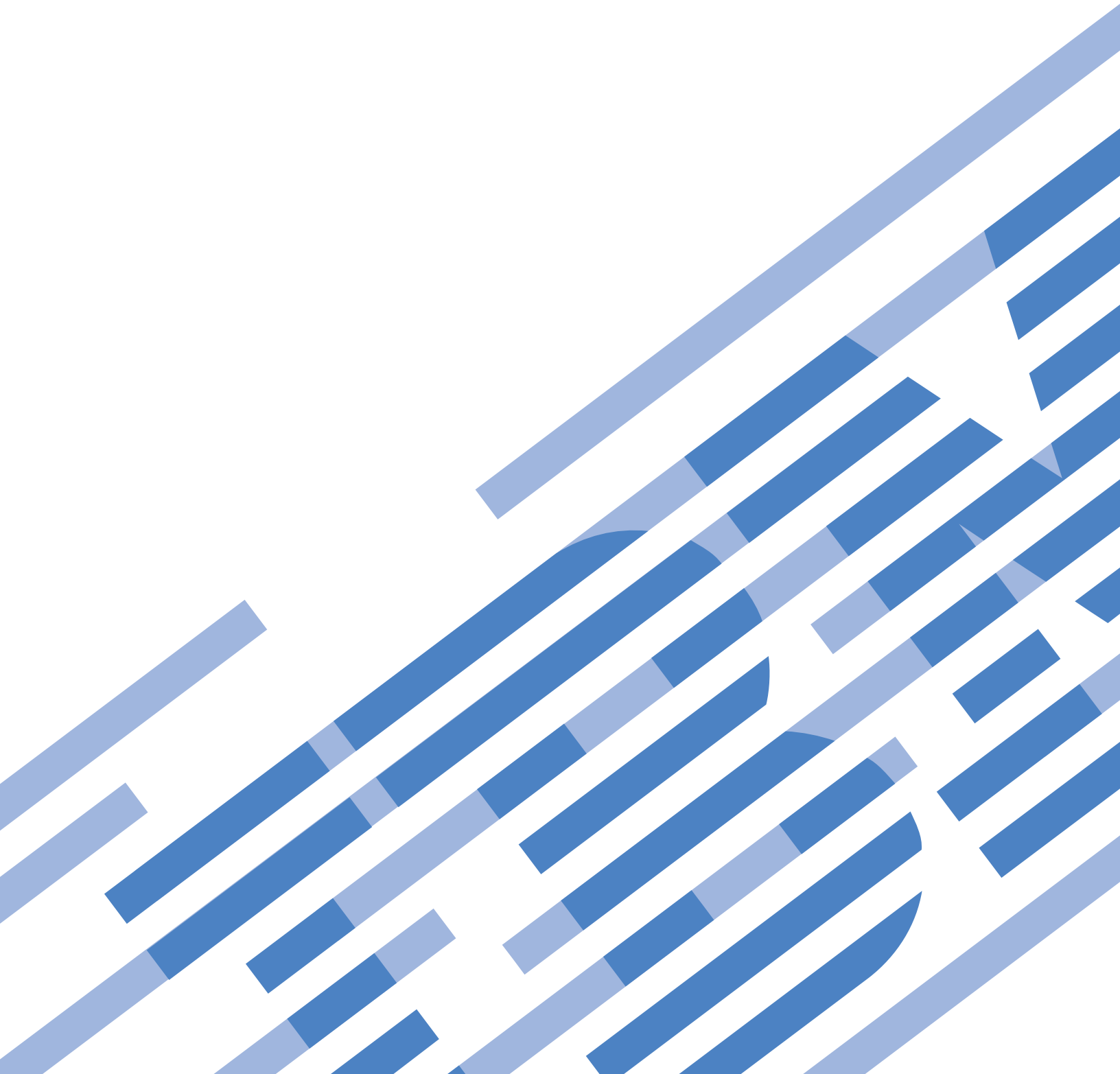

IBM Systems - iSeries

UNIX-Type -- Sockets APIs

Version 5 Release 4

IBM Systems - iSeries

UNIX-Type -- Sockets APIs

Version 5 Release 4

Note

Before using this information and the product it supports, be sure to read the information in "Notices," on page 339.

Sixth Edition (February 2006)

This edition applies to version 5, release 4, modification 0 of IBM i5/OS (product number 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Sockets APIs

The sockets APIs consist of functions, structures, and defined macros. The structures and defined macros are shipped as header files.

An important part of interprocess communications is to locate and construct network addresses. Many of the socket network APIs are inherently not threadsafe. Threadsafe APIs have been added to mirror the function provided by the non-threadsafe APIs. All threadsafe APIs follow the UNIX(R) convention of appending R to the API name denoting threadsafe.

There are two categories of sockets functions:

- "Sockets System Functions"
- "Sockets Network Functions" on page 196

For additional information, see:

- Sockets Programming
- "Debugging IP over SNA Configurations" on page 337

APIs

These are the APIs for this category.

Sockets System Functions

The system functions supported by the sockets APIs are:

- "accept()—Wait for Connection Request and Make Connection" on page 4 (Wait for an incoming connection and tie that connection to the application) is used to wait for connection requests.
- "accept_and_recv()—Wait for Connection Request and Receive the First Message That Was Sent" on page 8 (Wait for connection request and receive the first message that was sent.) is used to wait for an incoming connection request, receive the first message from the peer, and return the local and remote socket addresses associated with the connection.
- "bind()—Set Local Address for Socket" on page 13 (Set a local address for the socket) is used to associate a local address with a socket.
- "close()—Close File or Socket Descriptor" on page 19 (Close file descriptor) closes a descriptor, fildes.
- "connect()—Establish Connection or Destination Address" on page 22 (Bind a destination to a socket or set a connection) is used to establish a connection on a connection-oriented socket or establish the destination address on a connectionless socket.
- "fcntl()—Perform File Control Command" on page 28 (Perform file control command) performs various actions on open descriptors.
- "fstat()—Get File Information by Descriptor" on page 38 (Get file information by descriptor) gets status information about the file specified by the open file descriptor file_descriptor and stores the information in the area of memory indicated by the buf argument.
- "getdomainname()—Retrieve Domain Name" on page 42 (Retrieve domain name for the system) is used to retrieve the name of the domain from the system.
- "gethostid()—Retrieve Host ID" on page 43 (Retrieve host ID for the system) is used to retrieve a host ID's 32-bit IP address.

- "Rbind()—Set Remote Address for Socket" on page 106 (Establish remote bind) used to request that a SOCKS server allow an inbound connection request across a firewall.
- "read()—Read from Descriptor" on page 108 (Read from Descriptor) reads nbyte bytes of input into the memory area indicated by buf.
- "readv()—Read from Descriptor Using Multiple Buffers" on page 114 (Read from Descriptor Using Multiple Buffers) is used to receive data from a file or socket descriptor.
- "recv()—Receive Data" on page 119 (Receive data using a socket descriptor) is used to receive data through a socket.
- "recvfrom()—Receive Data" on page 122 (Receive data and remote address using a socket descriptor) is used to receive data through a connected or unconnected socket.
- "recvmsg()—Receive a Message Over a Socket" on page 126 (Receive data and remote address using a socket descriptor and multiple buffers (scatter read)) is used to receive data or descriptors or both through a connected or unconnected socket.
- "rexec()—Issue a Command on a Remote Host" on page 132 (Issue a command on a remote host) is used to open a connection to a remote host and send a user ID, password, and command to the remote host.
- "rexec_r()—Issue a Command on a Remote Host" on page 136 (Issue a command on a remote host) is used to open a connection to a remote host and send a user ID, password, and command to the remote host.
- "rexec_r_ts64()—Issue a Command on a Remote Host" on page 139 (Issue a command on a remote host) is used to open a connection to a remote host and send a user ID, password, and command to the remote host.
- "rexec_ts64()—Issue a Command on a Remote Host" on page 140 (Issue a command on a remote host) is used to open a connection to a remote host and send a user ID, password, and command to the remote host.
- "select()—Wait for Events on Multiple Sockets" on page 140 (Allow a single process to wait for connections on multiple sockets) is used to enable an application to multiplex I/O.
- "send()—Send Data" on page 143 (Send data using a socket descriptor) is used to send data through a connected socket.
- "send_file()—Send a File over a Socket Connection" on page 157 (Send a file over a socket connection) is used to send the contents of an open file over an existing socket connection.
- "send_file64()—Send a File over a Socket Connection" on page 161 (Send a file over a socket connection) is used to send the contents of an open file over an existing socket connection.
- "sendmsg()—Send a Message Over a Socket" on page 146 (Send data with a destination address using a socket descriptor and multiple buffers (gather write)) is used to send data or descriptors or both through a connected or unconnected socket.
- "sendto()—Send Data" on page 153 (Send data with a destination address using a socket descriptor) is used to send data through a connected or unconnected socket.
- "setdomainname()—Set Domain Name" on page 162 (Set domain name for the system) is used to set the name of the domain.
- "sethostid()—Set Host ID" on page 164 (Set Host ID) is used to set a host ID.
- "sethostname()—Set Host Name" on page 165 (Set host name for the system) is used to set the name of the host for a system.
- "setsockopt()—Set Socket Options" on page 167 (Allow an application to set characteristics of a socket (timeout, retransmission, buffer space)) is used to set socket options.
- "shutdown()—End Receiving and/or Sending of Data on Socket" on page 176 (End Receiving and/or Sending of Data on Socket) is used to disable reading, writing, or reading and writing on a socket.
- "socket()—Create Socket" on page 178 (Create a socket) is used to create an end point for communications.

- "socketpair()—Create a Pair of Sockets" on page 181 (Create a pair of sockets) is used to create a pair of unnamed, connected sockets in the AF_UNIX or AF_UNIX_CCSID address_family.
- "takedescriptor()—Receive Socket Access from Another Job" on page 183 (Receive the access rights to a descriptor) is used to obtain a descriptor in one OS/400 job which was passed from another OS/400 job by a givedescriptor().
- "write()—Write to Descriptor" on page 185 (Write to Descriptor) writes nbyte bytes from buf to the file or socket associated with file_descriptor.
- "writev()—Write to Descriptor Using Multiple Buffers" on page 192 (Write to Descriptor Using Multiple Buffers) is used to write data to a file or socket descriptor.

Note: These functions use header (include) files from the library QSYSINC, which is optionally installable. Make sure QSYSINC is installed on your system before using any of the functions.

Top | UNIX-Type APIs | APIs by category

---

accept()—Wait for Connection Request and Make Connection

BSD 4.3 Syntax

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

int accept(int socket_descriptor,
           struct sockaddr *address,
           int *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int accept(int socket_descriptor,
           struct sockaddr *address,
           socklen_t *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *accept()* function is used to wait for connection requests. *accept()* takes the first connection request on the queue of pending connection requests and creates a new socket to service the connection request.

accept() is used with connection-oriented socket types, such as SOCK_STREAM.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and struc(TM)tures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket on which to wait.

address

(Output) A pointer to a buffer of type **struct sockaddr** in which the address from which the connection request was received is stored. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

Note: See the usage notes about using different address families with **sockaddr_storage**.

address_length

(Input/output) This parameter is a value-result field. The caller passes a pointer to the length of the *address* parameter. On return from the call, *address_length* contains the actual length of the address from which the connection request was received.

Authorities

When the socket identified by the **socket_descriptor** is of type AF_INET and a connection indication request is received over an APPC device, the thread must have adequate authority. The thread must have retrieve, insert, delete, and update authority to the APPC device. When the thread does not have this level of authority, an *errno* of EACCES is returned.

Return Value

accept() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is a socket descriptor.

Error Conditions

When *accept()* fails, *errno* can be set to one of the following:

[EACCES]	Permission denied. A connection indication request was received on the socket referenced by the *socket_descriptor* parameter, but the process that issued the *accept()* did not have the appropriate privileges required to handle the request. The connection indication request is reset by the system.
[EBADF]	Descriptor not valid.

[ECONNABORTED] Connection ended abnormally.

An *accept()* was issued on a socket for which receives have been disallowed (due to a *shutdown()* call).

This also could be encountered if time elapsed since a successful *Rbind()* is greater than the margin allowed by the associated *SOCKS server*.

» An *accept()* was issued on a socket in blocking mode and one or more connections have been reset and there are no acceptable connections in the queue. This is only valid if socket option `SO_ACCEPTECONNABORTED` was enabled for the listening socket. «

[EFAULT] Bad address.

System detected an address which was not valid while attempting to access the *address* or *address_length* parameters.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *address_length* parameter is set to a value that is less than zero, and the *address* parameter is set to a value other than a NULL pointer.
- A *listen()* has not been issued against the socket referenced by the *socket_descriptor* parameter.

[EIO] Input/output error.

[EMFILE] Too many descriptions for this process.

[ENFILE] Too many descriptions in system.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

The *socket_descriptor* parameter references a socket that does not support the *accept()*. The *accept()* is only valid on sockets that are connection-oriented (for example, type of `SOCK_STREAM`).

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. If the *address* parameter is set to a NULL pointer or the *address_length* parameter points to an integer which has a value that is equal to zero, the address from which the connection request was received is not returned.
2. If the length of the address to be returned exceeds the length of the *address* parameter, the returned address is truncated.
3. The following are inherited by the descriptor returned by the accept() call:

- All socket options with a level of SOL_SOCKET.
- The status flags:
  - Blocking flag (set/reset either by the *ioctl()* call with the FIONBIO request or by the *fcntl()* call with the F_SETFL command and the status flag set to O_NONBLOCK).
  - Asynchronous flag (set/reset either by the *ioctl()* call with the FIOASYNC request or by the *fcntl()* call with the F_SETFL command and the status flag set to FASYNC).
- The process ID or process group ID that is to receive SIGIO or SIGURG signals (set/reset by either the *ioctl()* call with the FIOSETOWN or the SIOCSPGRP request, or by the *fcntl()* call with the F_SETOWN command).

4. Closing a socket causes any queued but unaccepted connection requests to be reset.
5. The structure **sockaddr** is a generic structure used for any address family but it is only 16 bytes long. The actual address returned for some address families may be much larger. You should declare storage for the address with the structure **sockaddr_storage**. This structure is large enough and aligned for any protocol-specific structure. It may then be cast as **sockaddr** structure for use on the APIs. The *ss_family* field of the **sockaddr_storage** will always align with the family field of any protocol-specific structure. The BSD 4.3 structure is:

```
#define _SS_MAXSIZE 304
#define _SS_ALIGNSIZE (sizeof (char*))
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof(sa_family_t))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))

struct sockaddr_storage {
    sa_family_t   ss_family;
    char         _ss_pad1[_SS_PAD1SIZE];
    char*        _ss_align;
    char         _ss_pad2[_SS_PAD2SIZE];
};
```

   The BSD 4.4/UNIX 98 compatible structure is:

```
#define _SS_MAXSIZE 304
#define _SS_ALIGNSIZE (sizeof (char*))
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))

struct sockaddr_storage {
    uint8_t       ss_len;
    sa_family_t   ss_family;
    char         _ss_pad1[_SS_PAD1SIZE];
    char*        _ss_align;
    char         _ss_pad2[_SS_PAD2SIZE];
};
```

6. If the socket is using an address family of AF_UNIX, the address (which is a path name) is returned in the default coded character set identifier (CCSID) currently in effect for the job.
7. If the socket is using an address family of AF_UNIX_CCSID, the output structure sockaddr_unc defines the format and coded character set identifier (CCSID) of the address (which is a path name).
8. If a successful *Rbind()* has been performed on the listening socket, then a new connection is not returned, but rather an inbound connection occurs on the same listening socket. The descriptor number returned is different, but it actually refers to the same connection referred to by the listening socket.
9. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *accept()* API is mapped to *qso_accept98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "bind()—Set Local Address for Socket" on page 13—Set Local Address for Socket
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- "listen()—Invite Incoming Connections Requests" on page 75—Invite Incoming Connections Requests

API introduced: V3R1

---

accept_and_recv()—Wait for Connection Request and Receive the First Message That Was Sent

BSD 4.3 Syntax

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

int accept_and_recv(int listen_socket_descriptor,
                    int *accept_socket_descriptor,
                    struct sockaddr *remote_address,
                    size_t *remote_address_length,
                    struct sockaddr *local_address,
                    size_t *local_address_length,
                    void *buffer,
                    size_t buffer_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int accept_and_recv(int listen_socket_descriptor,
                    int *accept_socket_descriptor,
                    struct sockaddr *remote_address,
                    socklen_t *remote_address_length,
                    struct sockaddr *local_address,
                    socklen_t *local_address_length,
                    void *buffer,
                    size_t buffer_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *accept_and_recv()* function is used to wait for an incoming connection request, receive the first message from the peer, and return the local and remote socket addresses associated with the connection.

accept_and_recv() is used with connection-oriented sockets that have an address family of `AF_INET` or `AF_INET6` and a socket type of `SOCK_STREAM`.

The *accept_and_recv()* API is a combination of the *accept()*, *getsockname()*, and *recv()* socket APIs. Socket applications that use these three APIs can obtain improved performance by using *accept_and_recv()*.

There are two versions of the API, as shown above. The base i5/OS(TM) API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

listen_socket_descriptor

(Input) The descriptor of the socket on which to wait. This parameter specifies the socket that has issued a successful call to *listen()*.

accept_socket_descriptor

(Input/Output) A pointer to an integer that specifies the socket descriptor on which to accept the incoming connection. This socket must not be bound or connected. The use of this parameter lets the application reuse the accepting socket.

If a pointer to a value of -1 is passed in for this parameter, a new descriptor in the process's descriptor table will be allocated for incoming connection. The socket descriptor for a new connection will be returned to the application by this parameter. It is recommended that a value of -1 be used on the first call to *accept_and_recv()*. See the "Usage Notes" on page 11 for additional information.

remote_address

(Output) A pointer to a buffer of type **struct sockaddr** in which the address from which the connection request was received is stored. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

Note: See the usage notes about using different address families with **sockaddr_storage**.

remote_address_length

(Input/Output) This parameter is a value-result field. The caller passes a pointer to the length of the *remote_address* parameter. On return from the call, *remote_address_length* contains the actual length of the address from which the connection request was received.

local_address

(Output) A pointer to a buffer of type **struct sockaddr** in which the local address over which the connection request was received is stored. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

Note: See the usage notes about using different address families with **sockaddr_storage**.

local_address_length
(Input/Output) This parameter is a value-result field. The caller passes a pointer to the length of the *local_address* parameter. On return from the call, *local_address_length* contains the actual length of the local address over which the connection request was received.

buffer (Output) The pointer to the buffer in which the data that is to be read is stored. If a NULL pointer is passed in for this parameter, the receive operation is not performed and the *accept_and_recv()* function completes when the incoming connection is received.

buffer_length
(Input) The length in bytes of the buffer pointed to by the *buffer* parameter.

Authorities

If IP over SNA is being used, *CHANGE authority to the APPC device is required.

Return Value

accept_and_recv() returns an integer. Possible values are:

- -1 (unsuccessful call)
- n (successful call), where n is the number of bytes received.

Error Conditions

When *accept_and_recv()* fails, *errno* can be set to one of the following:

[EACCES]	Permission denied. A connection indication request was received on the socket referenced by the *listen_socket_descriptor* parameter, but the process that issued the *accept_and_recv()* call did not have the appropriate privileges required to handle the request. The connection indication request is reset by the system.
[EBADF]	Descriptor not valid. Either the *listen_socket_descriptor* or the descriptor pointed to by the *accept_socket_descriptor* parameter is not a valid socket descriptor.
[ECONNABORTED]	Connection ended abnormally. An *accept_and_recv()* was issued on a socket for which receive operations have been disallowed (due to a *shutdown()* call).
[EFAULT]	Bad address. System detected an address that was not valid while attempting to access the *accept_socket_descriptor*, *remote_address*, *remote_address_length*, *local_address*, *local_address_length*, or *buffer* parameter.
[EINTR]	Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- A *listen()* has not been issued against the socket referenced by the *listen_socket_descriptor* parameter.
- The socket referenced by the *accept_socket_descriptor* parameter has been bound to a local address.
- The *accept_socket_descriptor* does not have the same address family and socket type as the *listen_socket_descriptor*.
- The *accept_socket_descriptor* parameter is set to a value that is less than -1.

[EIO] Input/output error.

[EISCONN] A connection has already been established.

[EMFILE] Too many descriptions for this process.

[ENFILE] Too many descriptions in system.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTSOCK] The specified descriptor does not reference a socket.

Either the *listen_socket_descriptor* or the descriptor pointed to by the *accept_socket_descriptor* parameter is not a valid socket descriptor.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *listen_socket_descriptor* parameter references a socket that does not support the *accept_and_recv()* function. The *accept_and_recv()* function is only valid on sockets that have an address family of `AF_INET` or `AF_INET6` and a socket type of `SOCK_STREAM`.
- The `O_NONBLOCK` option is set for the *listen_socket_descriptor* or the descriptor pointed to by the *accept_socket_descriptor* parameter. Non-blocking is not supported for *accept_and_recv()*.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The *accept_and_recv()* function is only valid on sockets that have an address family of AF_INET or AF_INET6 and a socket type of SOCK_STREAM. If the *listen_socket_descriptor* does not have the correct address family and socket type, -1 is returned and the *errno* value is set to EOPNOTSUPP.
2. Non-blocking mode is not supported for this function. If O_NONBLOCK is set on the *listen_socket_descriptor* parameter or on the descriptor pointed to by the *accept_socket_descriptor* parameter, -1 is returned and the *errno* value is set to EOPNOTSUPP.
3. If the *remote_address* parameter is set to a NULL pointer, the address from which the connection request was received is not returned. If the length of the remote address to be returned exceeds the length that was specified by the *remote_address_length* parameter, the returned address will be truncated.

4. If the *local_address* parameter is set to a NULL pointer, the local address to which the socket is bound is not returned. If the length of the local address to be returned exceeds the length that was specified by the *local_address_length* parameter, the returned address will be truncated.
5. If the *buffer* parameter is set to a NULL pointer or the *buffer_length* parameter is set to value of 0, the receive operation is not performed and the *accept_and_recv()* function completes when the incoming connection is received.
6. If a pointer to a value of -1 is passed in for the *accept_socket_descriptor* parameter, the following attributes are inherited by the socket descriptor that is returned by the *accept_and_recv()* call:
   - All socket options with a level of SOL_SOCKET.
   - The status flags:
     – Asynchronous flag (set or reset either by the *ioctl()* call with the FIOASYNC request or by the *fcntl()* call with the F_SETFL command and the status flag set to FASYNC).
   - The process ID or process group ID that is to receive SIGIO or SIGURG signals (set or reset by either the *ioctl()* call with the FIOSETOWN or the SIOCSPGRP request, or by the *fcntl()* call with the F_SETOWN command).
7. The *accept_and_recv()* function allows an application to reuse an existing socket descriptor. If a socket descriptor is specified for the *accept_socket_descriptor* parameter, it must not be bound or connected and it must have the same address family and socket type as the *listen_socket_descriptor*. The socket descriptor that is passed in for the *accept_socket_descriptor* parameter can be obtained by either calling *socket()* or by specifying the SF_REUSE flag on the *flags* parameter of the *send_file()* function.

   If an application specifies a pointer to an unbound and unconnected socket descriptor for the *accept_socket_descriptor* parameter that is the same address family and socket type as the *listen_socket_descriptor*, the *accept_and_recv()* function will try to use the *accept_socket_descriptor* for the incoming connection. If the *accept_socket_descriptor* cannot be used for the incoming connection, the descriptor for that socket will be closed and a new socket will be created for the incoming connection. The new socket may have a different descriptor number associated with it. This means that the value that is returned by the *accept_socket_descriptor* parameter may not be the same value that was specified by the application when the *accept_and_recv()* function was called.

   The ability to reuse an existing socket is not supported on all platforms. Therefore, it is recommended that a pointer to a value of -1 be passed in for the *accept_socket_descriptor* parameter. If socket reuse is not supported and the *send_file()* API is called with the *flags* parameter set to SF_REUSE, the socket connection will be closed and the socket descriptor will be set to -1 by the *send_file()* API. If socket reuse is supported, then the connection will be closed and the socket descriptor will be reset so that it can be used again. Regardless of whether socket reuse is supported or not, the application can pass its socket descriptor variable into the *accept_and_recv()* function as the *accept_socket_descriptor* parameter.
8. The structure **sockaddr** is a generic structure used for any address family but it is only 16 bytes long. The actual address returned for some address families may be much larger. You should declare storage for the address with the structure **sockaddr_storage**. This structure is large enough and aligned for any protocol-specific structure. It may then be cast as **sockaddr** structure for use on the APIs. The *ss_family* field of the **sockaddr_storage** will always align with the family field of any protocol-specific structure.

   The BSD 4.3 structure is:

```
#define _SS_MAXSIZE 304
#define _SS_ALIGNSIZE (sizeof (char*))
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof(sa_family_t))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))

struct sockaddr_storage {
    sa_family_t   ss_family;
    char         _ss_pad1[_SS_PAD1SIZE];
    char*        _ss_align;
    char         _ss_pad2[_SS_PAD2SIZE];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
#define _SS_MAXSIZE 304
#define _SS_ALIGNSIZE (sizeof (char*))
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))

struct sockaddr_storage {
    uint8_t       ss_len;
    sa_family_t   ss_family;
    char         _ss_pad1[_SS_PAD1SIZE];
    char*        _ss_align;
    char         _ss_pad2[_SS_PAD2SIZE];
};
```

9. To take full advantage of the performance improvement offered by the *accept_and_recv()* API, a multiple accept server model needs to be used by the application. In this model the server will do a *socket()*, *bind()*, and *listen()* as currently is done. The server will then give the listening socket to multiple jobs or threads. Each job or thread will then call *accept_and_recv()* using the same listening socket. When a connection request comes in, only one of the jobs or threads would wake up.
10. If a successful *Rbind()* has been performed on the listening socket, then a new connection is not returned, but rather an inbound connection occurs on the same listening socket. The descriptor number returned is different, but it actually refers to the same connection referred to by the listening socket.
11. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *accept_and_recv()* API is mapped to *qso_accept_and_recv98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “accept()—Wait for Connection Request and Make Connection” on page 4—Wait for Connection Request and Make Connection
- “getsockname()—Retrieve Local Address of Socket” on page 49—Retrieve Local Address of Socket
- “recv()—Receive Data” on page 119—Receive Data
- “send_file()—Send a File over a Socket Connection” on page 157—Send a File over a Socket Connection

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

---

bind()—Set Local Address for Socket

BSD 4.3 Syntax

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

int bind(int socket_descriptor,
         struct sockaddr *local_address,
         int address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int bind(int socket_descriptor,
         const struct sockaddr *local_address,
         socklen_t address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *bind()* function is used to associate a local address with a socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket that is to be bound.

local_address
(Input) A pointer to a buffer of type **struct sockaddr** that contains the local address to which the socket is to be bound. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

address_length
(Input) The length of the *local_address*.

Authorities

- When the address type of the socket identified by the **socket_descriptor** is AF_INET, the thread must have retrieve, insert, delete, and update authority to the port specified by the **local_address** field. When the thread does not have this level of authority, an *errno* of EACCES is returned.
- When the address type of the socket identified by the **socket_descriptor** is AF_INET and is running IP over SNA, the thread must have retrieve, insert, delete, and update authority to the APPC device. When the thread does not have this level of authority, an *errno* of EACCES is returned.

Return Value

bind() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When a *bind()* fails, *errno* can be set to one of the following:

[EACCES] Permission denied.

The process does not have the appropriate privileges to bind *local_address* to the socket pointed to by *socket_descriptor* (for example, if *socket_descriptor* is a socket with an address family of `AF_INET`, and the **sockaddr_in** structure (pointed to by *local_address*) specified a port that was restricted for use).

[EADDRINUSE] Address already in use.

This error code indicates one of the following:

- The *socket_descriptor* points to a socket with an address family of `AF_INET`, and the address specified in the **sockaddr_in** structure (pointed to by *local_address*) has already been assigned to another socket.
- The *socket_descriptor* points to a socket with an address family of `AF_INET6`, and the address specified in the **sockaddr_in6** structure (pointed to by *local_address*) has already been assigned to another socket.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX` or `AF_UNIX_CCSID`, and the address specified in the **sockaddr_un** or **sockaddr_unc** structure (pointed to by *local_address*) has already been assigned to another socket.

[EADDRNOTAVAIL] Address not available. This error code indicates one of the following:

- The *socket_descriptor* points to a socket with an address family of `AF_INET`, and the IP address specified in the **sockaddr_in** structure (pointed to by *local_address*) is not one defined by the local interfaces.
- The *socket_descriptor* points to a socket with an address family of `AF_INET6`, and the IP address specified in the **sockaddr_in6** structure (pointed to by *local_address*) is not one defined by the local interfaces.

[EAFNOSUPPORT] The type of socket is not supported in this protocol family.

The address family specified in the address structure pointed to by *local_address* parameter cannot be used with the socket pointed to by the *socket_descriptor* parameter.

[EBADF] Descriptor not valid.

» *[EDESTADDRREQ]* The *local_address* parameter is a null pointer.

This error code is only returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family. «

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *local_address* parameter.

[EINVAL] Parameter not valid. This error code indicates one of the following:

- The *address_length* parameter specifies a length that is negative or is not valid for the address family.
- The socket referenced by *socket_descriptor* is not a socket of type SOCK_RAW and is already bound to an address.
- The local address pointed to by the *local_address* parameter specified an address that was not valid.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and the CCSID specified in *sunc_qlg* in the **sockaddr_unc** structure (pointed to by *local_address*) cannot be converted to the current default CCSID for integrated file system path names.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and there was an incomplete character or shift state sequence at the end of *sunc_path* in the **sockaddr_unc** structure (pointed to by *local_address*).
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and the **sockaddr_unc** structure (pointed to by *local_address*) was not valid:
  - The sunc_format was not set to SO_UNC_DEFAULT or SO_UNC_USE_QLG.
  - The sunc_zero was not initialized to zeros.
  - The sunc_format field was set to SO_UNC_USE_QLG and the sunc_qlg structure was not valid:
    - The path type was less than 0 or greater than 3.
    - The path length was less than 0 or out of bounds. For example, a single-byte path name was greater than 126 bytes or a double-byte path name was greater than 252 bytes.
    - A reserved field was not initialized to zeros.

[EIO] Input/output error.

[ELOOP] A loop exists in symbolic links encountered during pathname resolution.

This error code is only returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENAMETOOLONG] File name too long.

This error code is only returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOENT] No such file or directory.

This error code is only returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOSYS] Function not implemented.

This error code is only returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` or `AF_UNIX_CCSID` address family.

[ENOTDIR] Not a directory.

This error code is only returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EUNKNOWN] Unknown system state.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. For sockets that use an address family of AF_UNIX or AF_UNIX_CCSID, the following is applicable:

   - The process must have the following types of permission:
     – Create permission to the directory in which the entry is to be created.
     – Search permission along all the components of the path.

     Also, processes trying to establish a connection with the *connect()* must have write access to the entry that is created.
   - For AF_UNIX, the path name is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job. For AF_UNIX_CCSID, the path name is assumed to be in the format and CCSID specified in the **sockaddr_unc** (pointed to by *local_address*).
   - When the socket is no longer needed, the caller should remove the file system entry that was created by the *bind()* using the *unlink()* or Qp0lunlink() system function.
2. For sockets that use an address family of AF_INET, the following is applicable:

   - The internet address structure **sockaddr_in** requires a 2-byte port number and a 32-bit IP address. You can have the system automatically select a port number by setting the port number to 0.

     The BSD 4.3 structure is:

     ```
     struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
     };
     ```

     The BSD 4.4/UNIX 98 compatible structure is:

     ```
     typedef uchar   sa_family_t;

     struct sockaddr_in {
        uint8_t        sin_len;
        sa_family_t    sin_family;
        u_short        sin_port;
        struct in_addr sin_addr;
        char           sin_zero[8];
     };
     ```

     The BSD 4.4 *sin_len* field is the length of the address. The *sin_family* is the address family (always AF_INET for TCP and UDP), *sin_port* is the port number, and *sin_addr* is the internet address. The *sin_zero* field is reserved and must be hex zeros.
   - A wildcard address is provided (INADDR_ANY defined in **<netinet/in.h>**) that allows an application to receive messages directed to a specified port independent of the IP address that was specified. If a local IP address is specified, only data received on that IP address is made available. INADDR_ANY must be used to receive data from multiple local interface definitions.
3. For sockets that use an address family of AF_INET6, the following is applicable:

- The internet address structure **sockaddr_in6** requires a 2-byte port number and a 128-bit IP address. You can have the system automatically select a port number by setting the port number to 0.

  The BSD 4.3 structure is:

```
typedef unsigned short sa_family_t;
typedef unsigned short in_port_t;

struct sockaddr_in6 {
   sa_family_t     sin6_family;
   in_port_t       sin6_port;
   uint32_t        sin6_flowinfo;
   struct in6_addr sin6_addr;
   uint32_t        sin6_scope_id;
};
```

  The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;
typedef unsigned short in_port_t;

struct sockaddr_in6 {
   uint8_t         sin6_len;
   sa_family_t     sin6_family;
   in_port_t       sin6_port;
   uint32_t        sin6_flowinfo;
   struct in6_addr sin6_addr;
   uint32_t        sin6_scope_id;
};
```

  The BSD 4.4 *sin6_len* field is the length of the address. The *sin6_family* is the address family (AF_INET6 in this case), *sin6_port* is the port number, and *sin6_addr* is the internet address. The *sin6_flowinfo* field contains two pieces of information: the traffic class and the flow label. Note: This field is currently not supported and should be set to zero for upward compatibility. The *sin6_scope_id* field identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. Note: This field is currently not supported and should be set to zero for upward compatibility.

- A wildcard address is provided that allows an application to receive messages directed to a specified port independent of the IP address that was specified. Since the IPv6 address type is a structure (`struct in6_addr`), a symbolic constant can be used to initialize an IPv6 address variable, but cannot be used in an assignment. Therfore, the IPv6 wildcard address is provided in two forms as defined in **<netinet/in.h>**. The first version is a global variable named `in6addr_any`. This version is used similarly to the way applications use the `INADDR_ANY` in IPv4 as defined above and must be used for structure assignment. The other version is a symbolic constant named `IN6ADDR_ANY_INIT`. This version may be used to initialize an `in6_addr` structure. If a local IP address is specified, only data received on that IP address is made available. The wildcard address must be used to receive data from multiple local interface definitions.

4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *bind()* API is mapped to *qso_bind98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "connect()—Establish Connection or Destination Address" on page 22—Establish Connection or Destination Address

API introduced: V3R1

close()—Close File or Socket Descriptor

Syntax

```
#include <unistd.h>

int close(int fildes);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 20.

The **close()** function closes a descriptor, *fildes*. This frees the descriptor to be returned by future **open()** calls and other calls that create descriptors.

When the last open descriptor for a file is closed, the file itself is closed. If the link count of the file is zero at that time, the space occupied by the file is freed and the file becomes inaccessible.

close() unlocks (removes) all outstanding byte locks that a job has on the associated file.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO is discarded and internal storage used is returned to the system.

When *fildes* refers to a socket, **close()** closes the socket identified by the descriptor.

For information about the exit point that can be associated with **close()**, see Integrated File System Scan on Close Exit Programs.

Parameters

fildes (Input) The descriptor to be closed.

Authorities

No authorization is required. Authorization is verified during **open()**, **creat()**, or **socket()**.

Return Value

0 **close()** was successful.
-1 **close()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **close()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN]	
[EBADF]	
[EBADFID]	
[EBUSY]	
[EDAMAGE]	

Error condition	Additional information
[EDEADLK]	
[EINTR]	
[EINVAL]	
[EIO]	
[EJRNDAMAGE]	
[EJRNENTTOOLONG]	
[EJRNINACTIVE]	
[EJRNRCVSPC]	
[ENEWJRN]	
[ENEWJRNRCV]	
[ENOBUFS]	
[ENOSPC]	
[ENOSYS]	
[ENOTAVAIL]	
[ENOTSAFE]	
[ESCANFAILURE]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN]	

Additionally, if interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ETIMEDOUT]	
[EUNATCH]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [EBADF] when *fildes* is a scan descriptor that was passed to one of the scan-related exit programs. See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.
2. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

- Where multiple threads exist in the job.
- The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

  - ″Root″ (/)
  - QOpenSys
  - User-defined
  - QNTC
  - QSYS.LIB
  - Independent ASP QSYS.LIB
  - QOPT
  - Network File System
  - QFileSvr.400

3. When a socket descriptor is closed, the system tries to send any queued data associated with the socket.

   - For AF_INET sockets, depending on whether the SO_LINGER socket option is set, queued data may be discarded.

     Note: For these sockets, the default value for the SO_LINGER socket option has the option flag set off (the system attempts to send any queued data with an infinite wait time).

4. A socket descriptor being shared among multiple processes is not closed until the process that issued the *close()* is the last process with access to the socket.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- creat()—Create or Rewrite File
- dup()—Duplicate Open File Descriptor
- dup2()—Duplicate Open File Descriptor to Another Descriptor
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- Integrated File System Scan on Close Exit Programs
- open()—Open File
- “setsockopt()—Set Socket Options” on page 167—Set Socket Options
- unlink()—Remove Link to File

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **close()**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

main() {
  int fd1, fd2;
  char out[20]="Test string",
       fn[]="test.file",
       in[20];
  short write_error;
```

```
  memset(in, 0x00, sizeof(in));

  write_error = 0;

  if ( (fd1 = creat(fn,S_IRWXU)) == -1)
    perror("creat() error");
  else if ( (fd2 = open(fn,O_RDWR)) == -1)
    perror("open() error");
  else {
    if (write(fd1, out, strlen(out)+1) == -1) {
      perror("write() error");
      write_error = 1;
    }
    close(fd1);
    if (!write_error) {
      if (read(fd2, in, sizeof(in)) == -1)
        perror("read() error");
      else printf("string read from file was: '%s'\n", in);
    }
    close(fd2);
  }
}
```

Output:

```
string read from file was: 'Test string'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

connect()—Establish Connection or Destination Address

BSD 4.3 Syntax

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

int connect(int socket_descriptor,
            struct sockaddr *destination_address,
            int address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int connect(int socket_descriptor,
            const struct sockaddr *destination_address,
            socklen_t address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *connect()* function is used to establish a connection on a connection-oriented socket or establish the destination address on a connectionless socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket that is to be connected.

destination_address
(Input) A pointer to a buffer of type **struct sockaddr** that contains the destination address to which the socket is to be bound. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

address_length
(Input) The length of the *destination_address*.

Authorities

When the address type of the socket identified by the **socket_descriptor** is AF_INET and is running IP over SNA, the thread must have retrieve, insert, delete, and update authority to the APPC device. When the thread does not have this level of authority, then an *errno* of EACCES is returned.

Return Value

connect() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When a *connect()* fails, *errno* can be set to one of the following. For additional debugging information, see "Debugging IP over SNA Configurations" on page 337.

[EACCES] Permission denied.

This error code indicates one of the following:

- The process does not have the appropriate privileges to connect to the address pointed to by the *destination_address* parameter.
- The socket pointed to by *socket_descriptor* is using a connection-oriented transport service, and the *destination_address* parameter specifies a TCP/IP limited broadcast address (internet address of all ones).

[EADDRINUSE] Address already in use.

This error code indicates one of the following:

- The *socket_descriptor* parameter points to a connection-oriented socket that has been bound to a local address that contained no wildcard values, and the *destination_address* parameter specified an address that matched the bound address.
- The *socket_descriptor* parameter points to a socket that has been bound to a local address that contained no wildcard values, and the *destination_address* parameter (also containing no wildcard values) specified an address that would have resulted in a connection with a non-unique association.

[EADDRNOTAVAIL] Address not available.

This error code is returned if the *socket_descriptor* parameter points to a socket with an address family of `AF_INET` or `AF_INET6` and either a port was not available or a route to the address specified by the *destination_address* parameter could not be found.

[EAFNOSUPPORT] The type of socket is not supported in this protocol family.

The address family specified in the address structure pointed to by *destination_address* parameter cannot be used with the socket pointed to by the *socket_descriptor* parameter.

[EALREADY] Operation already in progress.

A previous *connect()* function had already been issued for the socket pointed to by the *socket_descriptor* parameter, and has yet to be completed. This error code is returned only on sockets that use a connection-oriented transport service.

[EBADF] Descriptor not valid.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

This error occurs when there is no application that is bound to the address specified by the *destination_address* parameter.

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *destination_address* parameter.

[EHOSTUNREACH] A route to the remote host is not available.

This error code is returned on sockets that use the `AF_INET` and `AF_INET6` address families.

[EINPROGRESS] Operation in progress.

The *socket_descriptor* parameter points to a socket that is marked as nonblocking and the connection could not be completed immediately. This error code is returned only on sockets that use a connection-oriented transport service.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *address_length* parameter specifies a length that is negative or not valid for the address family.
- The AF_INET or AF_INET6 socket is of type SOCK_STREAM, and a previous *connect()* has already completed unsuccessfully. Only one connection attempt is allowed on a connection-oriented socket.

  Note: For sockets that have an address family of AF_UNIX, or AF_UNIX_CCSID, if a *connect()* fails, a subsequent *connect()* is allowed, even if the transport service being used is connection-oriented.
- *connect()* cannot be issued on the socket pointed to by the *socket_descriptor* parameter because the socket is using a connection-oriented transport service (with an address family of AF_INET or AF_INET6), and a *shutdown()* that disabled the sending of data was previously issued.
- The destination address pointed to by the *destination_address* parameter specified an address that was not valid.
- The *socket_descriptor* points to a socket with an address family of AF_UNIX_CCSID, and the CCSID specified in *sunc_qlg* in the **sockaddr_unc** structure (pointed to by *local_address*) cannot be converted to the current default CCSID for integrated file system path names.
- The *socket_descriptor* points to a socket with an address family of AF_UNIX_CCSID, and there was an incomplete character or shift state sequence at the end of *sunc_path* in the **sockaddr_unc** structure (pointed to by *local_address*).
- The *socket_descriptor* points to a socket with an address family of AF_UNIX_CCSID, and the **sockaddr_unc** structure (pointed to by *local_address*) was not valid:

  – The sunc_format was not set to SO_UNC_DEFAULT or SO_UNC_USE_QLG.
  – The sunc_zero was not initialized to zeros.
  – The sunc_format field was set to SO_UNC_USE_QLG and the sunc_qlg structure was not valid:

    - The path type was less than 0 or greater than 3.
    - The path length was less than 0 or out of bounds. For example, a single byte path name was greater than 126 bytes or a double byte path name was greater than 252 bytes.
    - A reserved field was not initialized to zeros.

[EIO] Input/output error.

[EISCONN] A connection has already been established.

This error code is returned only on sockets that use a connection-oriented transport service.

[ELOOP] A loop exists in symbolic links encountered during pathname resolution.

This error code is only returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.

[ENAMETOOLONG] File name too long.

This error code is only returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.

[ENETDOWN] The network is not currently available.

[ENETUNREACH]	Cannot reach the destination network. This error code is returned for sockets that use the AF_INET or AF_INET6 address families, the address specified by the *destination_address* parameter requires the use of a router, and the socket option SO_DONTROUTE is currently set on.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOENT]	No such file or directory. This error code is only returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.
[ENOSYS]	Function not implemented. This error code is only returned on sockets that use the AF_UNIX and AF_UNIX_CCSID address families.
[ENOTDIR]	Not a directory.
[ENOTSOCK]	The specified descriptor does not reference a socket. This error code is only returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.
[EOPNOTSUPP]	Operation not supported. *connect()* is not allowed on a passive socket (a socket for which a *listen()* has been done).
[EPROTOTYPE]	The socket type or protocols are not compatible. This error code is only returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.
[ETIMEDOUT]	A remote host did not respond within the timeout period. This error code is returned when connection establishment times out. No connection is established. A possible cause may be that the partner application is bound to the address specified by the *destination_address* parameter, but the partner application has not yet issued a *listen()*.
[EUNKNOWN]	Unknown system state.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.
[EPROTO]	An underlying protocol error has occurred.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. *connect()* establishes an end-to-end connection. It can only be issued once on sockets that have an address family of AF_INET or AF_INET6 and are of type SOCK_STREAM. (If the *connect()* fails to successfully establish the connection, you must close the socket and create a new socket if you wish to try to establish a connection again.) For sockets of other address families that are connection-oriented, you may simply try the *connect()* again to the same or to a new address. *connect()* can be issued on sockets of type SOCK_DGRAM and SOCK_RAW multiple times. Each time *connect()* is issued, it changes the destination address from which packets may be received and to which packets may be sent.

   Note: Issuing *connect()* on sockets of type SOCK_DGRAM and SOCK_RAW is not recommended because of dynamic route reassignment (picking a new route when a route that was previously used is no longer

available). When this reassignment occurs, the next packet from the partner program can be received from a different IP address than the address your application specified on the *connect()*. This results in the data being discarded.

2. When a *connect()* is issued successfully on sockets with an address family of AF_INET or AF_INET6 and type of SOCK_DGRAM, errors relating to the unsuccessful delivery of outgoing packets may be received as *errno* values. For example, assume an application has issued the *connect()* for a *destination_address* at which no server is currently bound for the port specified in *destination_address*, and the application sends several packets to that *destination_address*. Eventually, one of the application output functions (for example, *send()*) will receive an error [ECONNREFUSED]. If the application had not issued the *connect()*, this diagnostic information would have been discarded.
3. A connectionless transport socket for which a *connect()* has been issued can be disconnected by either setting the *destination_address* parameter to NULL or setting the *address_length* parameter to zero, and issuing another *connect()*.
4. For sockets that use a connection-oriented transport service and an address family of AF_INET or AF_INET6 there is a notion of a directed connect. A **directed connect** allows two socket endpoints (socket A and socket B) to be connected without having a passive socket to accept an incoming connection request. The idea is for both sockets to bind to addresses. Socket A then issues a *connect()* specifying the address that socket B is bound to, and socket B issues a *connect()* specifying the address that socket A is bound to. At this point sockets A and B are connected, and data transfer between the sockets can now take place.
5. For sockets with an address family of AF_INET or AF_INET6, the following is applicable:
   - For sockets of type SOCK_STREAM or SOCK_DGRAM, a local port number is implicitly assigned to the socket if the *connect()* is issued without previously issuing a *bind()*.
6. For sockets with an address family of AF_INET, the following is applicable:
   - If the destination address has an IP address that is set to zero, the system selects an appropriate destination IP address using the following algorithm:
     – If the socket is bound to an IP address of zero, a loopback address is used. If a loopback interface is not configured (or the associated interface is not active), the address of the next available interface that is active is used. Otherwise, the destination IP address is not changed (and results in an error on the *connect()*).
     – If the socket is bound to a nonzero IP address, then the IP address that the socket is bound to is used.
   - If the destination address has an internet IP address that is set to INADDR_BROADCAST (hex 0xFFFFFFFF), the system selects an appropriate destination IP address using the following algorithm:
     – If the socket is bound to an IP address of zero and:
       - It is using a connectionless transport service, then the first active interface found that supports broadcast frames is used by the networking software.
       - It is using a connection-oriented transport service, an error is returned ([EACCES]).
     – If the socket is bound to a nonzero IP address and is using a connectionless transport service and:
       - The address that the socket is bound to denotes an interface that supports broadcast frames (for example, not a loopback address), then the limited broadcast address of the IP address that the socket is bound to is used.
       - The address that the socket is bound to is a loopback address, an error is returned ([EINVAL]).
     – If the socket is bound to a nonzero IP address and it is using a connection-oriented transport service, an error is returned ([EACCES]).
7. For sockets with an address family of AF_UNIX or AF_UNIX_CCSID, the following is applicable:
   - There is no implicit binding of an address to the socket. The socket is unnamed if the *connect()* is issued without previously issuing a *bind()*.

- The process must have write access to the destination address and search permission along all the components of the path.
- For AF_UNIX, the path name is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job. For AF_UNIX_CCSID, the path name is assumed to be in the format and coded character set identifier (CCSID) specified in the **sockaddr_unc** (pointed to by *local_address*).

8. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *connect()* API is mapped to *qso_connect98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "fcntl()—Perform File Control Command"—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- "bind()—Set Local Address for Socket" on page 13—Set Local Address for Socket
- "accept()—Wait for Connection Request and Make Connection" on page 4—Wait for Connection Request and Make Connection
- "sendto()—Send Data" on page 153—Send Data
- "sendmsg()—Send a Message Over a Socket" on page 146—Send Data or Descriptors or Both

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

fcntl()—Perform File Control Command

Syntax

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int descriptor,
          int command,
          ...)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 36.

The **fcntl()** function performs various actions on open descriptors, such as obtaining or changing the attributes of a file or socket descriptor.

Parameters

descriptor
(Input) The descriptor on which the control command is to be performed, such as having its attributes retrieved or changed.

command
(Input) The command that is to be performed on the *descriptor*.

...
(Input) A variable number of optional parameters that is dependent on the *command*. Only some of the commands use this parameter.

The *fcntl()* commands that are supported are:

F_DUPFD Duplicates the descriptor. A third **int** argument must be specified. **fcntl()** returns the lowest descriptor greater than or equal to this third argument that is not already associated with an open file. This descriptor refers to the same object as *descriptor* and shares any locks. If the original descriptor was opened in text mode, data conversion is also done on the duplicated descriptor. The FD_CLOEXEC flag that is associated with the new descriptor is cleared.

F_GETFD Obtains the descriptor flags for *descriptor*. **fcntl()** returns these flags as its result. For a list of supported file descriptor flags, see "Flags." Descriptor flags are associated with a single descriptor and do not affect other descriptors that refer to the same object.

F_GETFL Obtains the » open « flags for *descriptor*. **fcntl()** returns these flags as its result. For a list of » the open « flags, see Using the oflag Parameter in **open()**.

F_GETLK Obtains locking information for an object. You must specify a third argument of type `struct flock *`. See "File Locking" on page 30 for details. **fcntl()** returns 0 if it successfully obtains the locking information. When you develop in C-based languages and the function is compiled with the _LARGE_FILES macro defined, F_GETLK is mapped to the F_GETLK64 symbol.

F_GETLK64 Obtains locking information for a large file. You must specify a third argument of type `struct flock64 *`. See "File Locking" on page 30 for details. **fcntl()** returns 0 if it successfully obtains the locking information. When you develop in C-based languages, it is necessary to compile the function with the _LARGE_FILE_API macro defined to use this symbol.

F_GETOWN Returns the process ID or process group ID that is set to receive the `SIGIO` (I/O is possible on a descriptor) and `SIGURG` (urgent condition is present) signals. For more information, see Signal APIs.

F_SETFD Sets the descriptor flags for *descriptor*. You must specify a third *int* argument, which gives the new file descriptor flag settings (see "Flags"). If any other bits in the third argument are set, **fcntl()** fails with the [EINVAL] error. **fcntl()** returns 0 if it successfully sets the flags. Descriptor flags are associated with a single descriptor and do not affect other descriptors that refer to the same object.

F_SETFL Sets status flags for the descriptor. You must specify a third *int* argument, giving the new file status flag settings (see "Flags"). **fcntl()** does not change the file access mode, and file access bits in the third argument are ignored. All other oflag values that are valid on the **open()** API are also ignored. If any other bits in the third argument are set, **fcntl()** fails with the [EINVAL] error. **fcntl()** returns `0` if it successfully sets the flags.

F_SETLK Sets or clears a file segment lock. You must specify a third argument of type `struct flock *`. See "File Locking" on page 30 for details. **fcntl()** returns 0 if it successfully clears the lock. When you develop in C-based languages and the function is compiled with the _LARGE_FILES macro defined, F_SETLK is mapped to the F_SETLK64 symbol.

F_SETLK64 Sets or clears a file segment lock for a large file. You must specify a third argument of type `struct flock64 *`. See "File Locking" on page 30 for details. **fcntl()** returns 0 if it successfully clears the lock. When you develop in C-based languages, it is necessary to compile the function with the _LARGE_FILE_API macro defined to use this symbol.

F_SETLKW Sets or clears a file segment lock; however, if a shared or exclusive lock is blocked by other locks, **fcntl()** waits until the request can be satisfied. You must specify a third argument of type `struct flock *`. See "File Locking" on page 30 for details. When you develop in C-based languages and the function is compiled with the _LARGE_FILES macro defined, F_SETLKW is mapped to the F_SETLKW64 symbol.

F_SETLKW64 Sets or clears a file segment lock on a large file; however, if a shared or exclusive lock is blocked by other locks, **fcntl()** waits until the request can be satisfied. See "File Locking" on page 30 for details. You must specify a third argument of type `struct flock64 *`. When you develop in C-based languages, it is necessary to compile the function with the _LARGE_FILE_API macro defined to use this symbol.

F_SETOWN Sets the process ID or process group ID that is to receive the `SIGIO` and `SIGURG` signals. For more information, see Signal APIs.

Flags

There are several types of flags associated with each open object. Flags for an object are represented by symbols defined in the <**fcntl.h** header file. The following *file status* flags can be associated with an object:

FASYNC The SIGIO signal is sent to the process when it is possible to do I/O.
» This function will fail with error code [EINVAL] when *fildes* is for an object other than a socket.
«

FNDELAY This flag is defined to be equivalent to O_NDELAY.

O_APPEND Append mode. If this flag is 1, every write operation on the file begins at the end of the file.

O_DSYNC Synchronous update - data only. If this flag is 1, all file data is written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, and **write()**.

O_NDELAY This flag is defined to be equivalent to O_NONBLOCK.

O_NONBLOCK Non-blocking mode. If this flag is 1, read or write operations on the file will not cause the thread to block. This file status flag applies only to pipe, FIFO, and socket descriptors.

O_RSYNC Synchronous read. If this flag is 1, read operations to the file will be performed synchronously. This flag is used in combination with O_SYNC or O_DSYNC. When O_RSYNC and O_SYNC are set, all file data and file attributes are written to permanent storage before the read operation returns. When O_RSYNC and O_DSYNC are set, all file data is written to permanent storage before the read operation returns.

O_SYNC Synchronous update. If this flag is 1, all file data and file attributes relative to the I/O operation are written to permanent storage before the update operation returns. Update operations include, but are not limited to, the following: **ftruncate()**, **open()** with O_TRUNC, and **write()**.

The following *file access mode* flags can be associated with a file:

O_RDONLY The file is opened for reading only.

O_RDWR The file is opened for reading and writing.

O_WRONLY The file is opened for writing only.

A mask can be used to extract flags:

O_ACCMODE Extracts file access mode flags.

The following *descriptor* flags can be associated with a descriptor:

FD_CLOEXEC Controls descriptor inheritance during **spawn()** and **spawnp()** when simple inheritance is being used, as follows:

- If the FD_CLOEXEC flag is zero, the descriptor is inherited by the child process that is created by the **spawn()** or **spawnp()**API.

  Note: Descriptors that are created as a result of the **opendir()** API (to implement open directory streams) are not inherited, regardless of the value of the FD_CLOEXEC flag.
- If the FD_CLOEXEC flag is set, the descriptor is not inherited by the child process that is created by the **spawn()** or **spawnp()** API.

Refer to spawn()—Spawn Process and spawnp()—Spawn Process with Path for additional information about FD_CLOEXEC.

File Locking

A local or remote job can use **fcntl()** to lock out other local or remote jobs from a part of a file. By locking out other jobs, the job can read or write to that part of the file without interference from others. File locking can ensure data integrity when several jobs have a file accessed concurrently. For more

information about remote locking, see information about the network lock manager and the network status monitor in the Network File System Support book.

All locks obtained using **fcntl()** are advisory only. Jobs can use advisory locks to inform each other that they want to protect parts of a file, but advisory locks do not prevent input and output on the locked parts. If a job has appropriate permissions on a file, it can perform whatever I/O it chooses, regardless of what advisory locks are set. Therefore, advisory locking is only a convention, and it works only when all jobs respect the convention.

Another type of lock, called a mandatory lock, can be set by a remote personal computer application. Mandatory locks restrict I/O on the locked parts. A read fails when reading a part that is locked with a mandatory write lock. A write fails when writing a part that is locked with a mandatory read or mandatory write lock.

Two different structures are used to control locking operations: `struct flock` and `struct flock64` (both defined in the <**fcntl.h** header file). You can use `struct flock64` with the F_GETLK64, F_SETLK64, and F_SETLKW64 commands to control locks on large files (files greater than 2GB minus 1 byte). The `struct flock` structure has the following members:

short	l_type	Indicates the type of lock, as indicated by one of the following symbols (defined in the <**fcntl.h**> header file): *F_RDLCK* Indicates a *read lock*; also called a *shared lock*. When a job has a read lock, no other job can obtain write locks for that part of the file. More than one job can have a read lock on the same part of a file simultaneously. To establish a read lock, a job must have the file accessed for reading. *F_WRLCK* Indicates a *write lock*; also called an *exclusive lock*. When a job has a write lock, no other job can obtain a read lock or write lock on the same part or an overlapping part of that file. A job cannot put a write lock on part of a file if another job already has a read lock on an overlapping part of the file. To establish a write lock, a job must have accessed the file for writing. *F_UNLCK* Unlocks a lock that was set previously.
short	l_whence	One of three symbols used in determining the part of the file that is affected by this lock. These symbols are defined in the <**unistd.h**> header file and are the same as symbols used by **lseek()**: *SEEK_CUR* The current file offset in the file. *SEEK_END* The end of the file. *SEEK_SET* The start of the file.
off_t	l_start	Gives a byte offset used to identify the part of the file that is affected by this lock. If `l_start` is negative, it is handled as an unsigned value. The part of the file affected by the lock begins at this offset from the location given by `l_whence`. For example, if `l_whence` is SEEK_SET and `l_start` is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.
off_t	l_len	Gives the size of the locked part of the file, in bytes. If the size is negative, it is treated as an unsigned value. If `l_len` is zero, the locked part of the file begins at the position specified by `l_whence` and `l_start`, and extends to the end of the file. Together, `l_whence`, `l_start`, and `l_len` are used to describe the part of the file that is affected by this lock.

pid_t	l_pid	Specifies the job ID of the job that holds the lock. This is an output field used only with F_GETLK actions.
void	*l_reserved0	Reserved. Must be set to NULL.
void	*l_reserved1	Reserved. Must be set to NULL.

When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, the `struct flock` data type will be mapped to a `struct flock64` data type. To use the `struct flock64` data type explicitly, it is necessary to compile the function with _LARGE_FILE_API defined.

The `struct flock64` structure has the following members:

short	l_type	Indicates the type of lock, as indicated by one of the following symbols (defined in the <**fcntl.h** header file): *F_RDLCK* Indicates a *read lock*; also called a *shared lock*. When a job has a read lock, no other job can obtain write locks for that part of the file. More than one job can have a read lock on the same part of a file simultaneously. To establish a read lock, a job must have the file accessed for reading. *F_WRLCK* Indicates a *write lock*; also called an *exclusive lock*. When a job has a write lock, no other job can obtain a read lock or write lock on the same part or an overlapping part of that file. A job cannot put a write lock on part of a file if another job already has a read lock on an overlapping part of the file. To establish a write lock, a job must have accessed the file for writing. *F_UNLCK* Unlocks a lock that was set previously.
short	l_whence	One of three symbols used in determining the part of the file that is affected by this lock. These symbols are defined in the <**unistd.h**> header file and are the same as symbols used by **lseek()**: *SEEK_CUR* The current file offset in the file. *SEEK_END* The end of the file. *SEEK_SET* The start of the file.
char	l_reserved2[4]	Reserved field
off64_t	l_start	Gives a byte offset used to identify the part of the file that is affected by this lock. `l_start` is handled as a signed value. The part of the file affected by the lock begins at this offset from the location given by `l_whence`. For example, if `l_whence` is SEEK_SET and `l_start` is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.
off64_t	l_len	Gives the size of the locked part of the file, in bytes. If the size is negative, the part of the file affected is l_start + l_len through l_start - 1. If `l_len` is zero, the locked part of the file begins at the position specified by `l_whence` and `l_start`, and extends to the end of the file. Together, `l_whence`, `l_start`, and `l_len` are used to describe the part of the file that is affected by this lock.
pid_t	l_pid	Specifies the job ID of the job that holds the lock. This is an output field used only with F_GETLK actions.
char	reserved3[4]	Reserved field.
void	*l_reserved0	Reserved. Must be set to NULL.
void	*l_reserved1	Reserved. Must be set to NULL.

You can set locks by specifying F_SETLK or F_SETLK64 as the *command* argument for **fcntl()**. Such a function call requires a third argument pointing to a `struct flock` structure (or `struct flock64` in the case of F_SETLK64), as in this example:

```
struct flock lock_it;
lock_it.l_type = F_RDLCK;
lock_it.l_whence = SEEK_SET;
lock_it.l_start = 0;
lock_it.l_len = 100;
fcntl(file_descriptor,F_SETLK,&lock_it);
```

This example sets up a flock structure describing a read lock on the first 100 bytes of a file, and then calls **fcntl()** to establish the lock. You can unlock this lock by setting `l_type` to F_UNLCK and making the same call. If an F_SETLK operation cannot set a lock, it returns immediately with an error saying that the lock cannot be set.

The F_SETLKW and F_SETLKW64 operations are similar to F_SETLK and F_SETLK64, except that they wait until the lock can be set. For example, if you want to establish an exclusive lock and some other job already has a lock established on an overlapping part of the file, **fcntl()** waits until the other process has removed its lock.

F_SETLKW and F_SETLKW64 operations can encounter *deadlocks* when job A is waiting for job B to unlock a region and job B is waiting for job A to unlock a different region. If the system detects that an F_SETLKW or F_SETLKW64 might cause a deadlock, **fcntl()** fails with *errno* set to [EDEADLK].

With the F_SETLK64, F_SETLKW64, and F_GETLK64 operations, the maximum offset that can be specified is the largest value that can be held in an 8-byte, signed integer.

A job can determine locking information about a file by using F_GETLK and F_GETLK64 as the *command* argument for **fcntl()**. In this case, the call to **fcntl()** should specify a third argument pointing to a flock structure. The structure should describe the lock operation you want. When **fcntl()** returns, the structure indicated by the flock pointer is changed to show the first lock that would prevent the proposed lock operation from taking place. The returned structure shows the type of lock that is set, the part of the file that is locked, and the job ID of the job that holds the lock. In the returned structure:

- `l_whence` is always SEEK_SET.
- `l_start` gives the offset of the locked portion from the beginning of the file.
- `l_len` is the length of the locked portion.

If there are no locks that prevent the proposed lock operation, the returned structure has F_UNLCK in `l_type` and is otherwise unchanged.

If **fcntl()** attempts to operate on a large file (one larger than 2GB minus 1 byte) with the F_SETLK, F_GETLK, or FSETLKW commands, the API fails with [EOVERFLOW]. To work with large files, compile with the _LARGE_FILE_API macro defined (when you develop in C-based languages) and use the F_SETLK64, F_GETLK64, or FSETLKW64 commands. When you develop in C-based languages, it is also possible to work with large files by compiling the source with the _LARGE_FILES macro label defined. Note that the file must have been opened for large file access (either the **open64()** API was used or the **open()** API was used with the O_LARGEFILE flag defined in the oflag parameter).

An application that uses the F_SETLK or F_SETLKW commands may try to lock or unlock a file that has been extended beyond 2GB minus 1 byte by another application. If the value of l_len is set to 0 on the lock or unlock request, the byte range held or released will go to the end of the file rather than ending at offset 2GB minus 2.

An application that uses the F_SETLK or F_SETLKW commands also may try to lock or unlock a file that has been extended beyond offset 2GB minus 2 with l_len NOT set to 0. If this application attempts to lock or unlock the byte range up to offset 2GB minus 2 and l_len is not 0, the unlock request will unlock the file only up to offset 2GB minus 2 rather than to the end of the file.

A job can have several locks on a file at the same time, but only one type of lock can be set on a given byte. Therefore, if a job puts a new lock on a part of a file that it had locked previously, the job has only one lock on that part of the file. The type of the lock is the one specified in the most recent locking operation.

Locks can start and extend beyond the current end of a file, but cannot start or extend ahead of the beginning of a file.

All of the locks a job has on a file are removed when the job closes any descriptor that refers to the locked file.

The maximum starting offset that can be specified by using the fnctl() API is 2^{63} - 1, the largest number that can be represented by a signed 8-byte integer. Mandatory locks set by a personal computer application or by a user of the DosSetFileLocks64() API may lock a byte range that is greater than 2^{63} - 1.

An application that uses the F_SETLK64 or F_SETLKW64 commands can lock the offset range that is beyond 2^{63} - 1 by locking offset 2^{63} - 1. When offset 2^{63} - 1 is locked, it implicitly locks to the end of the file. The end of the file is the largest number than can be represented by an 8-byte unsigned integer or 2^{64} - 1. This implicit lock may inhibit the personal computer application from setting mandatory locks in the range not explicitly accessable by the fcntl() API.

Any lock set using the fcntl() API that locks offset 2^{63} - 1 will have a length of 0.

An application that uses the F_GETLK64 may encounter a mandatory lock set by a personal computer application, which locks a range of offsets greater than 2^{63} - 1. This lock conflict will have a starting offset equal to or less than 2^{63} - 1 and a length of 0.

Authorities

No authorization is required.

Return Value

value	**fcntl()** was successful. The value returned depends on the *command* that was specified.
-1	**fcntl()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fcntl()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN]	The process tried to lock with F_SETLK, but the lock is in conflict with a previously established lock.
[EBADF]	

Error condition	Additional information
[EBADFID]	
[EBADFUNC]	A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
[EBUSY]	
[EDAMAGE]	
[EDEADLK]	
[EFAULT]	
[EINVAL]	
[EIO]	
[EMFILE]	
[ENOLCK]	
[ENOMEM]	
[ENOSYS]	
[ENOTAVAIL]	
[ENOTSAFE]	
[EOVERFLOW]	One of the values to be returned cannot be represented correctly. The command argument is F_GETLK, F_SETLK, or F_SETLKW and the offset of any byte in the requested segment cannot be represented correctly in a variable of type off_t (the offset is greater than 2GB minus 1 byte).
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ETIMEDOUT]	
[EUNATCH]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – "Root" (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400
2. If F_DUPFD is specified as the **fcntl()** command, this function will fail with error code [EBADF] when *fildes* is a scan descriptor that was passed to one of the scan-related exit programs. See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.
3. If the **fcntl()** command is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ENOTSUP] if F_SETLK, F_SETLK64, F_SETLKW or F_SETLKW64 is specified. See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.
4. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   The following **fcntl()** commands are not supported:
   - F_GETLK
   - F_SETLK
   - F_SETLKW

   Using any of these commands results in an [ENOSYS] error.
5. Network File System Differences

   Reading and writing to a file with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks. For more information about remote locking, see information about the network lock manager and the network status monitor in the Network File System Support book.
6. QNetWare File System Differences

   F_GETLK and F_SETLKW are not supported. F_RDLCK and F_WRLCK are ignored. All locks prevent reading and writing. Advisory locks are not supported. All locks are mandatory locks. Locking a file that is opened more than once in the same job with the same access mode is not supported, and its result is undefined.
7. This function will fail with the [EOVERFLOW] error if the command is F_GETLK, F_SETLK, or F_SETLKW and the offset or the length exceeds offset 2 GB minus 2.

8. When you develop in C-based languages and an application is compiled with the _LARGE_FILES macro defined, the `struct flock` data type will be mapped to a `struct flock64` data type. To use the `struct flock64` data type explicitly, it is necessary to compile the function with the _LARGE_FILE_API defined.
9. In several cases, similar function can be obtained by using *ioctl()*.

Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- The <**fcntl.h**> file (see Header Files for UNIX-Type Functions)
- "close()—Close File or Socket Descriptor" on page 19—Close File or Socket Descriptor
- dup()—Duplicate Open File Descriptor
- dup2()—Duplicate Open File Descriptor to Another Descriptor
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- lseek()—Set File Read/Write Offset
- open()—Open File
- spawn()—Spawn Process
- spawnp()—Spawn Process with Path
- Network File System Support  book

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses **fcntl()**:

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

int main()
{
  int flags;
  int append_flag;
  int nonblock_flag;
  int access_mode;
  int file_descriptor; /* File Descriptor */
  char *text1 = "abcdefghij";
  char *text2 = "0123456789";
  char read_buffer[25];

  memset(read_buffer, '\0', 25);

  /* create a new file */
  file_descriptor = creat("testfile",S_IRWXU);
  write(file_descriptor, text1, 10);
  close(file_descriptor);

  /* open the file with read/write access */
  file_descriptor = open("testfile", O_RDWR);
  read(file_descriptor, read_buffer,24);
  printf("first read is \'%s\'\n",read_buffer);

  /* reset file pointer to the beginning of the file */
  lseek(file_descriptor, 0, SEEK_SET);
  /* set append flag to prevent overwriting existing text */
  fcntl(file_descriptor, F_SETFL, O_APPEND);
```

```
  write(file_descriptor, text2, 10);
  lseek(file_descriptor, 0, SEEK_SET);
  read(file_descriptor, read_buffer,24);
  printf("second read is \'%s\'\n",read_buffer);

  close(file_descriptor);
  unlink("testfile");

  return 0;
}
```

Output:

```
first read is 'abcdefghij'
second read is 'abcdefghij0123456789'
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

fstat()—Get File Information by Descriptor

Syntax

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

int fstat(int descriptor,
          struct stat *buffer)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 40.

The **fstat()** function gets status information about the object specified by the open descriptor *descriptor* and stores the information in the area of memory indicated by the *buffer* argument. The status information is returned in a `stat` structure, as defined in the <**sys/stat.h**> header file.

Parameters

descriptor
(Input) The descriptor for which information is to be retrieved.

buffer (Output) A pointer to a buffer of type **struct stat** in which the information is returned. The structure pointed to by the *buffer* parameter is described in stat()— Get File Information.

The *st_mode*, *st_dev*, and *st_blksize* fields are the only fields set for socket descriptors. The *st_mode* field is set to a value that indicates the descriptor is a socket descriptor, the *st_dev* field is set to -1, and the *st_blksize* field is set to an optimal value determined by the system.

Authorities

No authorization is required.

Return Value

0 **fstat()** was successful. The information is returned in *buffer*.
-1 **fstat()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **fstat()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN]	
[EBADF]	
[EBADFID]	
[EBADFUNC]	A given descriptor or directory pointer is not valid for this operation. The specified descriptor is incorrect, or does not refer to an open object.
[EBUSY]	
[EDAMAGE]	
[EFAULT]	
[EINVAL]	This error code may be returned when the underlying object represented by the descriptor is unable to fill the **stat** structure (for example, if the function was issued against a socket descriptor that had its connection reset).
[EIO]	
[ENOBUFS]	
[ENOSYSRSC]	
[ENOTAVAIL]	
[ENOTSAFE]	
[EOVERFLOW]	The specified file exists and its size is too large to be represented in the structure pointed to by *buffer* (the file is larger than 2GB minus 1 byte).
[EPERM]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNATCH]	
[EUNKNOWN]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ETIMEDOUT]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when both of the following conditions occur:
   - Where multiple threads exist in the job.
   - The object this function is operating on resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - "Root" (/)
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - Independent ASP QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
2. Sockets-Specific Notes
   - The field *st_mode* can be inspected using the S_ISSOCK macro (defined in **<sys/stat.h>**) to determine if the descriptor is pointing to a socket descriptor.
   - For socket descriptors, use the send buffer size (this is the value returned for *st_blksize*) for the length parameter on your input and output functions. This can improve performance.

     Note: IBM reserves the right to change the calculation of the optimal send size.
3. QOPT File System Differences

   The value for st_atime will always be zero. The value for st_ctime will always be the creation date and time of the file or directory.

   The user, group, and other mode bits are always on for an object that exists on a volume not formatted in Universal Disk Format (UDF).

   fstat() on /QOPT will always return 2,147,483,647 for size fields.

   fstat() on optical volumes will return the volume capacity or 2,147,483,647, whichever is smaller.

   The file access time is not changed.
4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)
5. QNetWare File System Differences

   The QNetWare file system does not fully support mode bits. See the Netware on iSeries topic for more information.

6. » QFileSvr.400 File System Differences
   The value of st_vfs will always be 0 for remote objects accessed via QFileSvr.400.«
7. This function will fail with the [EOVERFLOW] error if the specified file exists and its size is too large to be represented in the structure pointed to by *buffer* (the file is larger than 2GB minus 1 byte).
8. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to **fstat64()**. Note that the type of the *buffer* parameter, `struct stat *`, also will be mapped to type `struct stat64 *`. See stat64() for more information on this structure.
9. » If a descriptor for a pipe or socket is passed to this function, the value of st_vfs will be 0. Therefore, information about these objects' corresponding file system cannot be obtained using the QP0L_RETRIEVE_MOUNTED_FILE_SYSTEMS option of QP0LFLOP()—Perform file system operation. «

Related Information

- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- The <**sys/stat.h**> file (see Header Files for UNIX-Type Functions)
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- fstat64()—Get File Information by Descriptor (Large File Enabled)
- lstat()—Get File or Link Information
- open()—Open File
- "socket()—Create Socket" on page 178—Create Socket
- stat()—Get File Information
- stat64()—Get File Information (Large File Enabled)
- » QP0LFLOP()—Perform file system operation«

Example

See Code disclaimer information for information pertaining to code examples.

The following example gets status information:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <time.h>

main() {
  char fn[]="temp.file";
  struct stat info;
  int file_descriptor;

  if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (fstat(file_descriptor, &info) != 0)
      perror("fstat() error");
    else {
      puts("fstat() returned:");
      printf("  inode:   %d\n",   (int) info.st_ino);
      printf(" dev id:   %d\n",   (int) info.st_dev);
      printf("   mode:   %08x\n",       info.st_mode);
      printf("  links:   %d\n",         info.st_nlink);
      printf("    uid:   %d\n",   (int) info.st_uid);
      printf("    gid:   %d\n",   (int) info.st_gid);
    }
```

```
    close(file_descriptor);
    unlink(fn);
  }
}
```

Output: Note that the output may vary from system to system.

```
fstat() returned:
  inode:   3057
 dev id:   1
   mode:   03000080
  links:   1
    uid:   137
    gid:   500
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getdomainname()—Retrieve Domain Name

Syntax

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

int getdomainname(char *name,
                  int length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *getdomainname()* function is used to retrieve the name of the domain from the system.

Parameters

name (Output) The **name** parameter can be one of the following:

- The pointer to a character array where the domain name is to be stored. The domain name is NULL-terminated unless the length of the domain name exceeds the length of the *name* parameter. In that case the domain name is truncated to the size of the *name* parameter.
- A NULL string when a *sethostname()* has not been previously issued since the last initial program load.

length (Input) The length of the *name* parameter. Maximum length of domain names is 255.

Authorities

None.

Return Value

getdomainname() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *getdomainname()* fails, *errno* can be set to one of the following:

[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *name* parameter.
[EINVAL]	Parameter not valid. The *length* parameter specifies a negative value.
[EIO]	Input/output error.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. When a process issues a *setdomainname()*, the name of the domain can be accessed by any process that issues a *getdomainname()*.
2. The name of the domain is reset to NULL when an initial program load is performed.

   Note: The domain name returned by this function is NOT related to the domain name of the domain name server that is configured using the Configure TCP/IP (CFGTCP) menu.
3. The domain name is returned in the default coded character set identifier (CCSID) currently in effect for the job.

Related Information

- "setdomainname()—Set Domain Name" on page 162—Set Domain Name

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

gethostid()—Retrieve Host ID

Syntax

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

int gethostid()
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
nbsp;Threadsafe: Yes

The *gethostid()* function is used to retrieve a host's ID.

Authorities

No authorization is required.

Return Value

gethostid() returns an integer. Possible values are:

- 0 when a *sethostid()* has not been issued previously since the last initial program load (IPL)
- n (successful), where n is the number specified on a previously issued *sethostid()* call

Usage Notes

1. When a process issues a *sethostid()*, the *host_id* can be accessed by any process that issues a *gethostid()*
2. The *host_id* is reset to zero when an initial program load is performed.
3. The *host_id* is a signed integer. Therefore, a -1 return value from the *gethostid()* may not indicate an error, but rather that a previous *sethostid()* was issued that specified a *host_id* of -1.
4. While many socket implementations refer to the *host_id* as the IP address of the machine, this is not necessarily the case. Many machines that support the TCP/IP protocol suite support multiple local IP addresses. The value contained in *host_id* is **not** used by TCP in any manner.

Related Information

- "sethostid()—Set Host ID" on page 164—Set Host ID Address
- "gethostname()—Retrieve Host Name"—Retrieve Host Name
- "sethostname()—Set Host Name" on page 165—Set Host Name

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

gethostname()—Retrieve Host Name

BSD 4.3 Syntax

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

int gethostname(char *name,
                int length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int gethostname(char *name,
                socklen_t length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *gethostname()* function is used to retrieve the name of the host from the system.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

name (Output) The pointer to a character array where the host name is to be stored. The host name is NULL-terminated unless the length of the host name exceeds the length of the *name* parameter, in which case the host name is truncated to the size of the *name* parameter.

length (Input) The length of the *name* parameter.

Authorities

No authorization is required.

Return Value

gethostname() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *gethostname()* fails, *errno* can be set to one of the following:

[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *name* parameter.
[EINVAL]	Parameter not valid. The *length* parameter specifies a negative value.
[EIO]	Input/output error.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. Maximum length of host names is defined by {MAXHOSTNAMELEN} (defined in **<sys/param.h>**).
2. When a process issues a *sethostname()*, the host name can be accessed by any process that issues a *gethostname()*.
3. On an initial program load, the host name is set to whatever was configured using the iSeries Navigator or option 12 (Change TCP/IP domain information) on the Configure TCP/IP (CFGTCP) menu. The local domain name is appended with the local host name and stored in system-wide storage. This combined name is the host name that can be retrieved by *gethostname()*. If the local host name and local domain name are not set, the host name is set to NULL.
4. The host name is returned in the default coded character set identifier (CCSID) currently in effect for the job.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *gethostname()* API is mapped to *qso_gethostname98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "sethostname()—Set Host Name" on page 165—Set Host Name
- "gethostid()—Retrieve Host ID" on page 43—Retrieve Host ID Address
- "sethostid()—Set Host ID" on page 164—Set Host ID Address

API introduced: V3R1

---

getpeername()—Retrieve Destination Address of Socket

BSD 4.3 Syntax

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

int getpeername(int socket_descriptor,
                struct sockaddr *destination_address,
                int *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int getpeername(int socket_descriptor,
                struct sockaddr *destination_address,
                socklen_t *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *getpeername()* function is used to retrieve the destination address to which the socket is connected.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket for which the destination address is to be retrieved.

destination_address
(Output) A pointer to a buffer of type **struct sockaddr** in which the destination address to which the socket connects is stored. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

Note: See the usage notes about using different address families with **sockaddr_storage**.

address_length
(I/O) This parameter is a value-result field. The caller passes a pointer to the length of the *destination_address* parameter. On return from the call, the *address_length* parameter contains the actual length of the destination address.

Authorities

No authorization is required.

Return Value

getpeername() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *getpeername()* fails, *errno* can be set to one of the following:

[EBADF]	Descriptor not valid.
[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *destination_address* or *address_length* parameters.
[EINVAL]	Parameter not valid. The *address_length* parameter specifies a negative value.
[EIO]	Input/output error.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOTCONN]	Requested operation requires a connection.
[ENOTSOCK]	The specified descriptor does not reference a socket.
[EUNKNOWN]	Unknown system state.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. *getpeername()* fails if issued against a socket for which a *connect()* has not been done.
2. For connection oriented sockets, *getpeername()* fails if both the write side and the read side have been closed through the use of one or more previous *shutdown()* functions.
3. If the length of the address to be returned exceeds the length of the *destination_address* parameter, the returned address is truncated.
4. The structure **sockaddr** is a generic structure used for any address family but it is only 16 bytes long. The actual address returned for some address families may be much larger. You should declare storage for the address with the structure **sockaddr_storage**. This structure is large enough and aligned for any protocol-specific structure. It may then be cast as **sockaddr** structure for use on the APIs. The *ss_family* field of the **sockaddr_storage** will always align with the family field of any protocol-specific structure.

   The BSD 4.3 structure is:

   ```
   #define _SS_MAXSIZE 304
   #define _SS_ALIGNSIZE (sizeof (char*))
   #define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof(sa_family_t))
   #define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(sa_family_t)+
                         _SS_PAD1SIZE + _SS_ALIGNSIZE))

   struct sockaddr_storage {
       sa_family_t   ss_family;
       char         _ss_pad1[_SS_PAD1SIZE];
       char*        _ss_align;
       char         _ss_pad2[_SS_PAD2SIZE];
   };
   ```

   The BSD 4.4/UNIX 98 compatible structure is:

   ```
   #define _SS_MAXSIZE 304
   #define _SS_ALIGNSIZE (sizeof (char*))
   #define _SS_PAD1SIZE (_SS_ALIGNSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)))
   #define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)+
                         _SS_PAD1SIZE + _SS_ALIGNSIZE))

   struct sockaddr_storage {
       uint8_t       ss_len;
       sa_family_t   ss_family;
       char         _ss_pad1[_SS_PAD1SIZE];
       char*        _ss_align;
       char         _ss_pad2[_SS_PAD2SIZE];
   };
   ```

5. When used with an address family of AF_UNIX or AF_UNIX_CCSID, *getpeername()* always returns the same path name that was specified on the *bind()* in the peer program. If the path name specified by the peer program was not a fully qualified path name, the output of *getpeername()* is meaningful only if your program knows what current directory was in effect for the peer program when it issued the *bind()*. For AF_UNIX, the path name is returned in the default coded character set identifier (CCSID) currently in effect for the job. For AF_UNIX_CCSID, the output structure sockaddr_unc defines the format and CCSID of the returned path name.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getpeername()* API is mapped to *qso_getpeername98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "accept()—Wait for Connection Request and Make Connection" on page 4—Wait for Connection Request and Make Connection
- "bind()—Set Local Address for Socket" on page 13—Set Local Address for Socket
- "connect()—Establish Connection or Destination Address" on page 22—Establish Connection or Destination Address

- "getsockname()—Retrieve Local Address of Socket"—Retrieve Local Address of Socket

API introduced: V3R1

---

getsockname()—Retrieve Local Address of Socket

BSD 4.3 Syntax

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

int getsockname(int socket_descriptor,
                struct sockaddr *local_address,
                int *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int getsockname(int socket_descriptor,
                struct sockaddr *local_address,
                socklen_t *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *getsockname()* function is used to retrieve the local address associated with the socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket for which the local address is to be retrieved.

local_address
(Output) A pointer to a buffer of type **struct sockaddr** in which the local address of the socket is stored. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
```

```
    uint8_t     sa_len;
    sa_family_t sa_family;
    char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

Note: See the usage notes about using different address families with **sockaddr_storage**.

address_length
(I/O) This parameter is a value-result field. The caller passes a pointer to the length of the *local_address* parameter. On return from the call, the *address_length* parameter contains the actual length of the local address.

Authorities

No authorization is required.

Return Value

getsockname() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *getsockname()* fails, *errno* can be set to one of the following:

[EBADF]	Descriptor not valid.
[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *local_address* or *address_length* parameters.
[EINVAL]	Parameter not valid. This error code indicates one of the following: • The *address_length* parameter specifies a negative value. • The socket specified by the *socket_descriptor* parameter is using a connection-oriented transport service and either the write-side has been shut down (with a *shutdown()*) or the connection has been reset.
[EIO]	Input/output error.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOTSOCK]	The specified descriptor does not reference a socket.
[EUNKNOWN]	Unknown system state.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. If the length of the address to be returned exceeds the length of the *local_address* parameter, the returned address will be truncated.
2. The structure **sockaddr** is a generic structure used for any address family but it is only 16 bytes long. The actual address returned for some address families may be much larger. You should declare storage for the address with the structure **sockaddr_storage**. This structure is large enough and aligned for any protocol-specific structure. It may then be cast as **sockaddr** structure for use on the APIs. The *ss_family* field of the **sockaddr_storage** will always align with the family field of any protocol-specific structure.

   The BSD 4.3 structure is:

   ```
   #define _SS_MAXSIZE 304
   #define _SS_ALIGNSIZE (sizeof (char*))
   #define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof(sa_family_t))
   #define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(sa_family_t)+
                         _SS_PAD1SIZE + _SS_ALIGNSIZE))

   struct sockaddr_storage {
       sa_family_t   ss_family;
       char         _ss_pad1[_SS_PAD1SIZE];
       char*        _ss_align;
       char         _ss_pad2[_SS_PAD2SIZE];
   };
   ```

   The BSD 4.4/UNIX 98 compatible structure is:

   ```
   #define _SS_MAXSIZE 304
   #define _SS_ALIGNSIZE (sizeof (char*))
   #define _SS_PAD1SIZE (_SS_ALIGNSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)))
   #define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)+
                         _SS_PAD1SIZE + _SS_ALIGNSIZE))

   struct sockaddr_storage {
       uint8_t       ss_len;
       sa_family_t   ss_family;
       char         _ss_pad1[_SS_PAD1SIZE];
       char*        _ss_align;
       char         _ss_pad2[_SS_PAD2SIZE];
   };
   ```

3. When used with an address family of `AF_UNIX` or `AF_UNIX_CCSID`, *getsockname()* always returns the same path name that was specified on a *bind()*. If the path name that was specified is not a fully qualified path name, the output of *getsockname()* is meaningful only if your program knows what current directory was in effect at the time of the *bind()*. For `AF_UNIX`, the path name is returned in the default coded character set identifier (CCSID) currently in effect for the job. For `AF_UNIX_CCSID`, the output structure sockaddr_unc defines the format and CCSID of the returned path name.
4. *getsockname()* produces different results, depending on the address family or type of the socket:

   - For address family of `AF_INET`:

     – If the *type* is `SOCK_STREAM` or `SOCK_DGRAM`, *getsockname()* will return 0 if issued before the *bind()*. The socket address that is returned has the IP address and port number fields set to zeros.
     – If the *type* is `SOCK_RAW`, *getsockname()* returns a -1 if issued before a *bind()*.
     – If the *type* is `SOCK_STREAM`, and an *Rbind()* has successfully completed, then the address returned is the *SOCKS server* address. See *Rbind()* for more information.

   - For address family of `AF_INET6`:

     – If the *type* is `SOCK_STREAM` or `SOCK_DGRAM`, *getsockname()* will return 0 if issued before the *bind()*. The socket address that is returned has the IP address and port number fields set to zeros.

- If the *type* is SOCK_RAW, *getsockname()* returns a -1 if issued before a *bind()*.
- For address family of AF_UNIX or AF_UNIX_CCSID, *getsockname()* returns 0 if issued before a *bind()*. The address length is 0. This is always the case for sockets created by *socketpair()*.

5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getsockname()* API is mapped to *qso_getsockname98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "bind()—Set Local Address for Socket" on page 13—Set Local Address for Socket
- "connect()—Establish Connection or Destination Address" on page 22—Establish Connection or Destination Address

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getsockopt()—Retrieve Information about Socket Options

BSD 4.3 Syntax

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

int getsockopt(int socket_descriptor,
               int level,
               int option_name,
               char *option_value,
               int *option_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int getsockopt(int socket_descriptor,
               int level,
               int option_name,
               void *option_value,
               socklen_t *option_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *getsockopt()* function is used to retrieve information about socket options.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket for which information is to be retrieved.

level (Input) Value indicating whether the request applies to the socket itself or to the underlying protocol being used. Supported values are:

IPPROTO_IP	Request applies to IP protocol layer.
IPPROTO_TCP	Request applies to TCP protocol layer.
SOL_SOCKET	Request applies to socket layer.
IPPROTO_IPV6	Request applies to IPv6 protocol layer.
IPPROTO_ICMPV6	Request applies to ICMPv6 protocol layer.

option_name

(Input) The option name for which information is to be retrieved. The following tables list the options supported, and for which level the option applies. Assume that the option is supported for all address families unless the option is described otherwise.

Note: Options directed to a specific protocol level are only supported by that protocol. An option that is directed to *level* SOL_SOCKET usually completes successfully. If the underlying protocol does not provide support for the option, the socket library retrieves one of the following:

- The default value for the option.
- The value previously set with a *setsockopt()*.

This provides compatibility with **Berkeley Software Distributions** implementations that also shield the application from protocols that do not support an option.

Socket Options That Apply to the IP Layer (IPPROTO_IP)

Option	Description
IP_OPTIONS	Determine what options are set in the IP header. This is only supported for sockets with an address family of AF_INET.
IP_TOS	Get Type Of Service (TOS) and Precedence in the IP header. This option is only supported for sockets with an address family of AF_INET.
IP_TTL	Get Time To Live (TTL) in the IP header. This option is only supported for sockets with an address family of AF_INET.
IP_MULTICAST_IF	Get interface over which outgoing multicast datagrams will be sent. An *option_value* parameter of type in_addr is used to retrieve the local IP address that is associated with the interface over which outgoing multicast datagrams will be sent. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_MULTICAST_TTL	Get Time To Live (TTL) from the IP header for outgoing multicast datagrams. An *option_value* parameter of type char is used into which a value between 0 and 255 is retrieved. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_DONTFRAG	Return the current Don't fragment flag setting in the IP header. A value of 0 indicates that it is reset. A value of 1 indicates that it is set. This option is supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW only.
IP_MULTICAST_LOOP	Determine the multicast looping mode. A non-zero value indicates that multicast datagrams sent by this system should also be delivered to this system as long as it is a member of the multicast group. If this option is not set, a copy of the datagram will not be delivered to the sending host. An *option_value* parameter of type char is used to retrieve the current setting. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_RECVLCLIFADDR	Determine if the local interface that a datagram was received will be returned. A value of 1 indicates the first 4 bytes of the reserved field of the sockaddr structure will contain the local interface. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM.

Socket Options That Apply to the TCP Layer (IPPROTO_TCP)

Option	Description
TCP_NODELAY	Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default TCP will follow the Nagle algorithm. To disable this behavior, applications can enable TCP_NODELAY to force TCP to always send data immediately. A non-zero *option_value* returned by getsockopt indicates TCP_NODELAY is enabled. For example, TCP_NODELAY should be used when there is an appliciation using TCP for a request/response. This option is only supported for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
TCP_MAXSEG	Determine TCP maximum segment size. This option is only supported for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

Socket Options That Apply to the Socket Layer (SOL_SOCKET)

Option	Description
SO_ACCEPTCONN	Reports whether socket listening is enabled. This option stores an int value. This is a boolean option.
SO_ACCEPTECONNABORTED	Determine if the listening socket will return ECONNABORTED when a connection on the listening backlog is reset prior to a blocking *accept()*. The option is only valid on a socket that has successfully issued the *listen()* call. The option has no effect on non-blocking sockets. This option is only used by sockets with an address family of AF_INET or AF_INET6.
SO_BROADCAST	Determine if messages can be sent to the broadcast address. This option is only supported for sockets with an address family of AF_INET and type SOCK_DGRAM or SOCK_RAW. The broadcast address can be determined by issuing an *ioctl()* specifying the SIOCGIFBRDADDR request.
SO_DEBUG	Determine if low level-debugging is active.
SO_DONTROUTE	Determine if the normal routing mechanism is being bypassed. This option is only supported by sockets with an address family of AF_INET or AF_INET6.
SO_ERROR	Return any pending errors in the socket. The value returned corresponds to the standard error codes defined in **<errno.h>**
SO_KEEPALIVE	Determine if the connection is being kept up by periodic transmissions. This option is only supported for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
SO_LINGER	Determine whether the system attempts to deliver any buffered data or if the system discards it when a *close()* is issued. For sockets that are using a connection-oriented transport service with an address family of AF_INET or AF_INET6, the default is off (which means that the system attempts to send any queued data, with an infinite wait-time).
SO_OOBINLINE	Determine if out-of-band data is received inline with normal data. This option is only supported for sockets with an address family of AF_INET or AF_INET6.
SO_RCVBUF	Determine the size of the receive buffer.
SO_RCVLOWAT	Determine the size of the receive low-water mark. This option is only supported for sockets with a type of SOCK_STREAM.
SO_RCVTIMEO	Determine the receive timeout value. This option is not supported unless _XOPEN_SOURCE is defined to be 520 or greater.

Option	Description
SO_REUSEADDR	Determine if the local socket address can be reused. This option is supported by sockets with an address family of AF_INET or AF_INET6 and a type of SOCK_STREAM or SOCK_DGRAM.
SO_SNDBUF	Determine the size of the send buffer.
SO_SNDLOWAT	Determine the size of the send low-water mark. This option is not supported.
SO_SNDTIMEO	Determine the send timeout value. This option is not supported unless _XOPEN_SOURCE is defined to be 520 or greater.
SO_TYPE	Determine the value for the socket *type*.
SO_USELOOPBACK	Determine if the loopback feature is being used. This option is not supported.

Socket Options That Apply to the IPv6 Layer (IPPROTO_IPV6)

Option	Description
IPV6_UNICAST_HOPS	Get the hop limit value that will be used for subsequent unicast packets sent by this socket. An *option_value* parameter of type int is used to retrieve the current setting. This option is only supported for sockets with an address family of AF_INET6.
IPV6_MULTICAST_IF	Get the interface over which outgoing multicast datagrams will be sent. An *option_value* parameter of type unsigned int is used to retrieve the interface index that is associated with the interface over which outgoing multicast datagrams will be sent. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_MULTICAST_HOPS	Get the hop limit value that will be used for subsequent multicast packets sent by this socket. An *option_value* parameter of type int is used to retrieve the current setting. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_MULTICAST_LOOP	Determine the multicast looping mode. A value of 1 (default), indicates that multicast datagrams sent by this system should also be delivered to this system as long as it is a member of the multicast group. If this option is 0, a copy of the datagram will not be delivered to the sending host. An *option_value* parameter of type unsigned int is used to retrieve the current setting. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_V6ONLY	Determine the AF_INET6 communication restrictions. A non-zero value indicates that this AF_INET6 socket is restricted to IPv6 communications only. This option stores an int value. This is a boolean option. By default this option is turned off. This option is only supported for sockets with an address family of AF_INET6.
IPV6_CHECKSUM	Determine if the kernel will calculate and insert a checksum for output and verify the received checksum on input, discarding the packet if the checksum is in error for this socket. An *option_value* parameter of type int is used to retrieve the current setting. If this option is -1 (the default), this socket option is disabled. A value of 0 or greater specifies an integer offset into the user data of where the checksum is located. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_RAW with a protocol other than IPPROTO_ICMPV6. The checksum is automatically computed for protocol IPPROTO_ICMPV6.
IPV6_DONTFRAG	Determine if the kernel will not implement the automatic insertion of a fragment header in the packet if the packet is too big for the path MTU. A value of 0 disables the option meaning the default of automatic insertion will be used. A non-zero value indicates that the option is set meaning the kernel will discard the packet instead of inserting the fragment header. This option is supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW only.

Socket Options That Apply to the ICMPv6 Layer (IPPROTO_ICMPV6)

Option	Description
ICMP6_FILTER	Determine the current ICMPv6 Type Filtering. An *option_value* parameter of type **struct icmp6_filter**, defined in **<netinet/icmp6.h>** is used to retrieve the current setting. The following macros, defined in **<netinet/icmp6.h>** can be used after retrieval of the type filtering structure to determine whether or not specific ICMPv6 message types will be passed to the application or be blocked: ICMP6_FILTER_WILLPASS and ICMP6_FILTER_WILLBLOCK. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_RAW with a protocol of IPPROTO_ICMPV6.

option_value

(Output) A pointer to the option value. Integer flags/values are returned by *getsockopt()* for all the socket options except for SO_LINGER , IP_OPTIONS , IP_MULTICAST_IF , IP_MULTICAST_TTL, IP_MULTICAST_LOOP, and ICMP6_FILTER.

The following options should be considered as set if a nonzero value for the *option_value* parameter is returned:

- SO_ACCEPTCONN
- SO_ACCEPTECONNABORTED
- SO_BROADCAST
- SO_DEBUG
- SO_DONTROUTE
- SO_KEEPALIVE
- SO_OOBINLINE
- SO_REUSEADDR
- SO_USELOOPBACK
- TCP_NODELAY
- IP_MULTICAST_LOOP
- IP_DONTFRAG
- IPV6_V6ONLY
- IPV6_MULTICAST_IF
- IPV6_MULTICAST_LOOP
- IPV6_DONTFRAG

For the SO_LINGER option, *option_value* is a pointer to where the structure **linger** is stored. The structure **linger** is defined in **<sys/socket.h>**.

```
struct linger {
  int        l_onoff;
  int        l_linger;
};
```

The *l_onoff* field determines if the linger option is set. A nonzero value indicates the linger option is set and is using the *l_linger* value. A zero value indicates that the option is not set. The *l_linger* field is the time to wait before any buffered data to be sent is discarded. The following occur on a *close()*:

- For AF_INET and AF_INET6 sockets:
  - If the *l_onoff* value is zero, the system attempts to send any buffered data with an infinite wait-time.

- If the *l_onoff* value is nonzero and the *l_linger* value is nonzero, the system attempts to send any buffered data for *l_linger* time. If *l_linger* time has elapsed and the data is still not successfully sent, it is discarded. When data is discarded, the remote program may receive a `[ECONNRESET]`.

- For `AF_INET` sockets over SNA:

  - If the *l_onoff* value is nonzero and the *l_linger* value is zero, the system waits indefinitely (no timer is implemented). Otherwise, if the *l_onoff* value is nonzero and the *l_linger* value is zero, the system discards any buffered data. When data is discarded, the remote program may receive a `[ECONNRESET]`.

Note: An application must implement an application level confirmation. Guaranteed receipt of data by the partner program is required. Setting `SO_LINGER` does not guarantee delivery.

For the `SO_RCVTIME` and `SO_SNDTIME` options, *option_value* is a pointer to where the structure **timeval** is stored. The structure **timeval** is defined in **<sys/time.h>**.

```
struct timeval {
  long  tv_sec;
  long  tv_usec;
};
```

For the `IP_OPTIONS` option, *option_value* is a pointer to storage in which data representing the IP options (as specified in RFC 791) is stored. *getsockopt()* returns the options in the following format:

IP address	IP options	...	IP options

IP address is a 4-byte IP address, and IP options identifies the IP options that were set using *setsockopt()*. If an IP option set using *setsockopt()* contained a source routing option (strict or loose), the first IP address in the source routing option list is removed. The IP options are adjusted accordingly. (For this adjustment, the length in the IP options portion is changed, and alignment is kept by adding no-operation option). The buffer is returned in the same format. The first 4 bytes are the IP address that was removed, and this is followed by the remaining IP options, if any. If the IP options portion does not contain a source routing option, the first 4 bytes are set to zero.

For the `IP_MULTICAST_IF` option, *option_value* is a pointer to storage in which the structure **in_addr**, defined in **<netinet/in.h>** as the following, will be stored:

```
struct in_addr {
   u_long  s_addr; /* IP address  */
 };
```

The **s_addr** field that is returned will be the local IP address that is associated with the interface over which outgoing multicast datagrams are being sent.

Notes:

1. For sockets that use a connection-oriented transport service, IP options that are set using *setsockopt()* are only used if they are set prior to a *connect()* being issued. After the connection is established, any IP options that the user sets are ignored.
2. If the IP options portion contains a source routing option, then the address in the source routing option overrides the destination address. The destination address may have been specified on an output operation (for example, on a *sendto()*) or on a *connect()*.
3. If a socket has a type of SOCK_RAW and a protocol of IPPROTO_RAW, any IP options set using *setsockopt()* are ignored (since the user must supply the IP header data on an output operation as part of the data that is being transmitted).
4. The structure **ip_opts** (defined in **<netinet/in.h>**) can be used to receive IP options.

option_length
(I/O) The length of the *option_value*. The *option_length* parameter must be initially set by the caller. *option_length* is changed on return to indicate the actual amount of storage used.

Note: For option values that are of type integer, the length of the *option_value* pointed to by the *option_length* parameter must be set to a value that is greater or equal to the size of an integer. If the length is not set correctly, a correct option value is not received.

Authorities

The user profile for the thread must have `*IOSYSCFG` special authority to set options when the *level* parameter specifies `IPPROTO_IP` and the *option_value* parameter is `IP_OPTIONS`.

Return Value

getsockopt() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *getsockopt()* fails, *errno* can be set to one of the following:

[EBADF] Descriptor not valid.

[ECONNABORTED] Connection ended abnormally.

This error code indicates that the transport provider ended the connection abnormally because of one of the following:

- The retransmission limit has been reached for data that was being sent on the socket.
- A protocol error was detected.

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *option_value* or *option_length* parameters.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *level* parameter specifies a level that is not supported. (except for when the socket has an address family of `AF_UNIX`, in which case `[ENOPROTOOPT]` is returned).
- The *option_name* parameter specifies a value that is not valid (except for when the level is `SOL_SOCKET` , in which case `[ENOPROTOOPT]` is returned).
- The *option_length* parameter points to an integer that has a negative value.

[EIO] Input/output error.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOPROTOOPT] The protocol does not support the specified option.

This error code indicates one of the following:

- The socket has an address family of `AF_UNIX` and the *level* parameter specified is not `SOL_SOCKET` .
- The *level* parameter specifies a level of `SOL_SOCKET` and the *option_name* parameter specifies a value that is not valid.

[ENOTCONN]	Requested operation requires a connection. This error code is only returned if the *level* parameter specifies a level other than SOL_SOCKET and the *socket_descriptor* parameter points to a socket that is using a connection-oriented transport service that has had its connection broken.
[ENOTSOCK]	The specified descriptor does not reference a socket.
[EPERM]	Operation not permitted. The executing user profile must have *IOSYSCFG special authority to get options when the *level* parameter specifies IPPROTO_IP and the *option_value* parameter is IP_OPTIONS .
[EUNKNOWN]	Unknown system state.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. Socket options are defined in **<sys/socket.h>**, IP options are defined in **<netinet/ip.h>** and **<netinet/in.h>**, TCP options are defined in **<netinet/tcp.h>**, IPv6 and ICMPv6 options are defined in **<netinet/in.h>**.
2. When a TCP connection is closed for a socket using the AF_INET or AF_INET6 address families, the port associated with that connection is not made available until twice the Maximum Segment Life (MSL) time in seconds has passed. The MSL time is approximately 2 minutes. The SO_REUSEADDR option allows a *bind()* to succeed when requesting a port that is being held during this time frame. This can be especially useful if a server is abruptly ended and restarted.

   Notes:

   a. For AF_INET and AF_INET6, SOCK_STREAM sockets, this option does **not** allow two servers to successfully issue a *bind()* requesting the same port number and local address combination. For AF_INET and AF_INET6, SOCK_DGRAM sockets, the SO_REUSEADDR option does allow multiple servers to successfully bind to the same port. When broadcast or multicast datagrams are received for a given port, each server that is bound to that port receives a copy of the datagram provided each server has enabled the SO_REUSEADDR option.

   b. This option does not affect unicast datagram delivery.
3. Issuing a *getsockopt()* with the SO_ERROR option results in the resetting of the SO_ERROR option to zero. Issuing another *getsockopt()* with the SO_ERROR option also returns a value of zero, assuming no errors occur on the socket. Other functions, when issued, also reset the SO_ERROR option to zero. These functions are:
   - *read()*, *readv()*, *recv()*, *recvmsg()*, *recvfrom()*
   - *connect()* (only when using a connectionless transport service)
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getsockopt()* API is mapped to *qso_getsockopt98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "setsockopt()—Set Socket Options" on page 167—Set Socket Options

API introduced: V3R1

givedescriptor()—Pass Descriptor Access to Another Job

Syntax

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

int givedescriptor(int descriptor,
                   char *target_job)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *givedescriptor()* function is used to pass a descriptor from one i5/OS job to another i5/OS job.

Parameters

descriptor
(Input) The descriptor that is to be passed to the target job.

target_job
(Input) A pointer to the internal job identifier of the target job that is to receive the descriptor referenced by the *descriptor* parameter.

Authorities

To give a descriptor, the source thread must be running under one of the following user profiles:

- A user profile that is the same as the job user identity of the target job
- A user profile that has all object (*ALLOBJ) special authority

The **job user identity** is the name of the user profile by which a job is known to other jobs. It is described in more detail in the Work Management topic.

Return Value

givedescriptor() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *givedescriptor()* fails, *errno* can be set to one of the following:

[EACCES] Permission denied.

The job does not have the appropriate privileges required to give the descriptor.

[EBADF] Descriptor not valid.

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *target_job* parameter.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *target_job* parameter points to data that is not valid.
- The *target_job* parameter refers to a job that is not active.

[EIO] Input/output error.

[EOPNOTSUPP] Operation not supported.

The underlying instance represented by the descriptor does not support passing access rights.

[EUNKNOWN] Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The information to specify in the *target_job* parameter can be obtained in the actual target job by using a work management API (for example, **QUSRJOBI**) to retrieve the *internal job identifier*.

   It is the responsibility of the application programmer to privately pass this information from the target job to the job that issues the *givedescriptor()*. One possible method that could be used to exchange this information is to use data queues.
2. The *target_job* does not have to be waiting on a *takedescriptor()* for the *givedescriptor()* to complete successfully.
3. If both the job in which the *givedescriptor()* is issued and the *target_job* end while a descriptor is in transit, the descriptor is reclaimed by the system, and the resource that it represents is closed.
4. For files and directories, *givedescriptor()* is only supported for objects in the Root, QOpenSys, User-defined file systems (UDFS), and Network File System (NFS).

Related Information

- "takedescriptor()—Receive Socket Access from Another Job" on page 183—Receive Socket Access from Another Job
- "sendmsg()—Send a Message Over a Socket" on page 146—Send Data or Descriptors or Both
- "recvmsg()—Receive a Message Over a Socket" on page 126—Receive Data or Descriptors or Both
- spawn()—Spawn Process

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

if_freenameindex()—Free Dynamic Memory Allocated by if_nameindex()

Syntax

```
#include  <net/if.h>

void if_freenameindex(struct if_nameindex *ptr);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***if_freenameindex()*** function frees the dynamic memory that was allocated by ***if_nameindex()***. After ***if_freenameindex()*** has been called, the application should not use the array of which *ptr* is the address.

Parameters

ptr **(Input)**
Array of *if_nameindex* structures. *ptr* MUST be a pointer that was returned by ***if_nameindex()***.

The structure **struct if_nameindex** is defined in **<net/if.h>**.

```
struct if_nameindex
{
  unsigned int   if_index;  /* 1, 2, ... */
  char          *if_name;   /* null terminated name */
};
```

Authorities

No authorization is required.

Return Value

None.

Error Conditions

errno can be set to:

[DEFAULT]
The memory pointed to by *ptr* can not be accessed.

Related Information

- "getsockopt()—Retrieve Information about Socket Options" on page 52—Retrieve Information about Socket Options
- "if_indextoname()—Map an Interface index to its Corresponding Name" on page 63Map an Interface Index to its Corresponding Name
- "if_nameindex()—Return All Interface Names and Indexes" on page 65—Return All Interface Names and Indexes
- "if_nametoindex()—Map an Interface Name to its Corresponding Index" on page 67—Map an Interface Name to its Corresponding Index
- "setsockopt()—Set Socket Options" on page 167—Set Socket Options

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **if_freenameindex()** is used:

```
#include <net/if.h>
#include <sys/types.h>
#include <errno.h>
```

```
void main()
{

  struct if_nameindex *interfaceArray = NULL;
  interfaceArray = if_nameindex(void);    /* retrieve the current interfaces */
  if (interfaceArray != NULL)
  {

    ...

    if_freenameindex(interfaceArray);     /* free the dynamic memory */
    interfaceArray = NULL;                /* prevent use after free  */
  }
  else
  {
     printf("if_nameindex() failed with errno =  %d %s \n",
            errno,strerror(errno));
     return;
  }

  ...

}
```

API introduced: V5R4

Top | UNIX-Type APIs | APIs by category

---

if_indextoname()—Map an Interface index to its Corresponding Name

Syntax

```
#include  <net/if.h>

char *if_indextoname(unsigned int ifindex, char *ifname);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***if_indextoname()*** function places the name of the interface with index *ifindex* into the buffer pointed at by *ifname*. When this function is called, *ifname* must point to a buffer of at least IFNAMSIZ bytes.

Parameters

ifindex **(Input)**
Interface index.

ifname **(Output)**
Pointer to a null terminated string containing the interface (line description) name returned.

Authorities

No authorization is required.

Return Value

if_indextoname() returns a pointer to a null terminated string containing the interface (line description) name. Possible values are:

- The value supplied in *ifname*. (successful)
- NULL (unsuccessful)

Error Conditions

When *if_indextoname()* fails, *errno* can be set to one of the following:

[ENXIO]
The specified interface index does not exist.

[EFAULT]
The buffer pointed to by *ifname* can not be accessed.

Usage Notes

1. The interface (line description) name stored at *ifname* will be returned in the default coded character set identifier (CCSID) currently in effect for the job. If this is not a single byte CCSID, then storage greater than IFNAMSIZ (16) bytes may be needed. 22 bytes is large enough for all CCSIDs.
2. It is important to note that the term ″Interface″ refers to the name on a line description (i.e. a physical interface) for this API. Other parts of the operating system, when refering to ″Interface,″ mean an IP address.

Related Information

- “getsockopt()—Retrieve Information about Socket Options” on page 52—Retrieve Information about Socket Options
- “if_freenameindex()—Free Dynamic Memory Allocated by if_nameindex()” on page 61—Free Memory Allocated by if_nameindex()
- “if_nameindex()—Return All Interface Names and Indexes” on page 65—Return All Interface Names and Indexes
- “if_nametoindex()—Map an Interface Name to its Corresponding Index” on page 67—Map an Interface Name to its Corresponding Index
- “setsockopt()—Set Socket Options” on page 167—Set Socket Options

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **if_indextoname()** is used:

```
#include <net/if.h>
#include <sys/types.h>
#include <errno.h>
ref
void main()
{
  char interfaceName[IFNAMSIZ];
  char *interface = if_indextoname(1, &interfaceName); /* retrieve the name of interface 1 */

  if (interface == NULL)
  {
     printf("if_indextoname() failed with errno =  %d %s \n",
            errno,strerror(errno));
     return;
  }
```

...

}

«

API introduced: V5R4

---

if_nameindex()—Return All Interface Names and Indexes

Syntax

```
#include  <net/if.h>

struct if_nameindex *if_nameindex(void);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***if_nameindex()*** function returns an array of *if_nameindex* structures, one structure per interface. The end of the array of structures is indicated by a structure with an *if_index* of 0 and an *if_name* of NULL.

Parameters

None.

Authorities

No authorization is required.

Return Value

if_nameindex() returns a pointer to an array of *if_nameindex* structures. Possible values are:

- Pointer to dynamically allocated memory for the array of *if_nameindex* structures. (successful). Note: the ***if_freenameindex()*** API should be used to free the array once it is no longer needed.
- NULL (unsuccessful)

The structure **struct if_nameindex** is defined in **<net/if.h>**.

```
struct if_nameindex {
unsigned int   if_index;  /* 1, 2, ... */
char          *if_name;   /* null terminated line name */
};
```

Error Conditions

When *if_nameindex()* fails, *errno* can be set to one of the following:

[ENXIO]
No interfaces names and indexes exist.

[ENOMEM]
No memory available for the *if_nameindex* array.

Usage Notes

1. The interface (line description) names stored at *if_name* will be returned in the default coded character set identifier (CCSID) currently in effect for the job.
2. It is important to note that the term ″Interface″ refers to the name on a line description (i.e. a physical interface) for this API. Other parts of the operating system, when refering to ″Interface,″ mean an IP address.
3. The array returned will contain only interfaces (line descriptions) that are IPv6 capable. The array may also contain a *LOOPBACK entry.

Related Information

- “getsockopt()—Retrieve Information about Socket Options” on page 52—Retrieve Information about Socket Options
- “if_freenameindex()—Free Dynamic Memory Allocated by if_nameindex()” on page 61—Free Memory Allocated by if_nameindex()
- “if_indextoname()—Map an Interface index to its Corresponding Name” on page 63—Map an Interface Index to its Corresponding Name
- “if_nametoindex()—Map an Interface Name to its Corresponding Index” on page 67—Map an Interface Name to its Corresponding Index
- “setsockopt()—Set Socket Options” on page 167—Set Socket Options

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **if_nameindex()** is used:

```
#include <net/if.h>
#include <sys/types.h>
#include <errno.h>

void main()
{

  struct if_nameindex *interfaceArray = NULL;
  interfaceArray = if_nameindex(void);    /* retrieve the current interfaces */
  if (interfaceArray != NULL)
  {
    ...

    if_freenameindex(interfaceArray);     /* free the dynamic memory */
    interfaceArray = NULL;                /* prevent use after free  */
  }
  else
  {
     printf("if_nameindex() failed with errno =  %d %s \n",
            errno,strerror(errno));
     return;
  }

  ...

}
```

API introduced: V5R4

if_nametoindex()—Map an Interface Name to its Corresponding Index

Syntax

```
#include  <net/if.h>

unsigned int if_nametoindex(const char *ifname);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***if_nametoindex()*** function returns the interface index corresponding to name *ifname*.

Parameters

ifname **(Input)**
Pointer to a null terminated string containing the interface (line description) name.

Authorities

No authorization is required.

Return Value

if_nametoindex() returns an unsigned integer. Possible values are:

- n (where n is the corresponding index value)
- 0 (unsuccessful)

Error Conditions

When *if_nametoindex()* fails, *errno* can be set to one of the following:

[ENXIO]
The specified interface name does not exist.

[EFAULT]
The buffer pointed to by *ifname* can not be accessed.

Usage Notes

1. The interface (line description) name found at *ifname* is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job.
2. It is important to note that the term ″Interface″ refers to the name on a line description (i.e. a physical interface) for this API. Other parts of the operating system, when refering to ″Interface,″ mean an IP address.

Related Information

- “getsockopt()—Retrieve Information about Socket Options” on page 52—Retrieve Information about Socket Options
- “if_freenameindex()—Free Dynamic Memory Allocated by if_nameindex()” on page 61—Free Memory Allocated by if_nameindex()
- “if_indextoname()—Map an Interface index to its Corresponding Name” on page 63—Map an Interface Index to its Corresponding Name
- “if_nameindex()—Return All Interface Names and Indexes” on page 65—Return All Interface Names and Indexes

- "setsockopt()—Set Socket Options" on page 167—Set Socket Options

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **if_nametoindex()** is used:

```
#include <net/if.h>
#include <sys/types.h>
#include <errno.h>

void main()
{

  unsigned int interfaceIndex = if_nametoindex("MYETH");
  if (interfaceIndex == 0)
  {
     printf("if_nametoindex() failed with errno =  %d %s \n",
            errno,strerror(errno));
     return;
  }

  ...

}
```

API introduced: V5R4

Top | UNIX-Type APIs | APIs by category

---

ioctl()—Perform I/O Control Request

Syntax

```
#include <sys/types.h>
#include <sys/ioctl.h>

int ioctl(int descriptor,
          unsigned long request,
   ...);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 75.

The **ioctl()** function performs control functions (requests) on a descriptor.

Parameters

descriptor
(Input) The descriptor on which the control request is to be performed.

request
(Input) The request that is to be performed on the *descriptor*.

... (Input) A variable number of optional parameters that are dependent on the request.

The *ioctl()* requests that are supported are:

FIOASYNC Set or clear the flag that allows the receipt of asynchronous I/O signals (SIGIO).

The third parameter represents a pointer to an integer flag. A nonzero value sets the socket to generate SIGIO signals, while a zero value sets the socket to not generate SIGIO signals. Note that before the SIGIO signals can be delivered, you must use either the FIOSETOWN or SIOCSPGRP *ioctl()* request, or the F_SETOWN *fcntl()* command to set a process ID or a process group ID to indicate what process or group of processes will receive the signal. Once conditioned to send SIGIO signals, a socket will generate SIGIO signals whenever certain significant conditions change on the socket. For example, SIGIO will be generated when normal data arrives on the socket, when out-of-band data arrives on the socket (in addition to the SIGURG signal), when an error occurs on the socket, or when end-of-file is received on the socket. It is also generated when a connection request is received on the socket (if it is a socket on which the *listen()* verb has been done). Also note that a socket can be set to generate the SIGIO signal by using the *fcntl()* command F_SETFL with a flag value specifying FASYNC.

FIOCCSID Return the coded character set ID (CCSID) associated with the open instance represented by the descriptor and the CCSID associated with the object. The third parameter represents a pointer to the structure Qp0lFIOCCSID, which is defined in <sys/ioctl.h>. This information may be necessary to correctly manipulate data read from or written to a file opened in another process.

If the open instance represented by the descriptor is in binary mode (the open() did not specify the O_TEXTDATA open flag), the open instance CCSID returned is equal to the object CCSID returned.

FIOGETOWN Get the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

The third parameter represents a pointer to a signed integer that will contain the process ID or the process group ID to which the socket is currently sending asynchronous signals such as SIGURG. A process ID is returned as a positive integer, and a process group ID is specified as a negative integer. A 0 value returned indicates that no asynchronous signals can be generated by the socket. A positive or a negative value indicates that the socket has been set to generate SIGURG signals.

FIONBIO Set or clear the nonblocking I/O flag (O_NONBLOCK oflag). The third parameter represents a pointer to an integer flag. A nonzero value sets the nonblocking I/O flag for the descriptor; a zero value clears the flag.

FIONREAD Return the number of bytes available to be read. The third parameter represents a pointer to an integer that is set to the number of bytes available to be read.

FIOSETOWN Set the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

The third parameter represents a pointer to a signed integer that contains the process ID or the process group ID to which the socket should send asynchronous signals such as SIGURG. A process ID is specified as a positive integer, and a process group ID is specified as a negative integer. Specifying a 0 value resets the socket such that no asynchronous signals are delivered. Specifying a process ID or a process group ID requests that sockets begin sending the SIGURG signal to the specified ID when out-of-band data arrives on the socket.

SIOCADDRT Add an entry to the interface routing table. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **rtentry**, which is defined in **<net/route.h>**:

```
struct rtentry [
   struct sockaddr rt_dst;
   struct sockaddr rt_mask;
   struct sockaddr rt_gateway;
   int rt_mtu;
   u_short rt_flags;
   u_short rt_refcnt;
   u_char rt_protocol;
   u_char rt_TOS;
   char rt_if[IFNAMSIZ];
];
```

The *rt_dst*, *rt_mask*, and *rt_gateway* fields are the route destination address, route address mask, and gateway address, respectively. *rt_mtu* is the maximum transfer unit associated with the route. *rt_flags* contains flags that give some information about a route (for example, whether the route was created dynamically, whether the route is usable, type of route, and so on). *rt_refcnt* indicates the number of references that exist to the route entry. *rt_protocol* indicates how the route entry was generated (for example, configuration, ICMP redirect, and so on). *rt_tos* is the type of service associated with the route. *rt_if* is a NULL-terminated string that represents the interface IP address in dotted decimal format that is associated with the route.

To add a route, the following fields must be set:

- *rt_dst*
- *rt_mask*
- *rt_gateway*
- *rt_tos*
- *rt_protocol*
- *rt_mtu* (Setting the *rt_mtu* value to zero essentially means use the MTU from the associated line description used when the route is bound to an IFC.)
- *rt_if* (*rt_if* can be set to the dotted decimal equivalent of INADDR_ANY, which is 0.)

In addition, the *rt_flags* bit flags can be set to the following:

- RTF_NOREBIND_IFC_FAIL if no rebinding of the route is to occur when the interface associated with the route fails.
- RTF_NOREBIND_IFC_ACTV if no rebinding is to occur when interfaces are activated or deactivated.

To delete a route, the following fields must be set:

- *rt_dst*
- *rt_mask*
- *rt_gateway*
- *rt_tos*
- *rt_protocol*

All other fields are ignored when adding or removing an entry.

SIOCATMARK Return the value indicating whether socket's read pointer is currently at the out-of-band mark.

The third parameter represents a pointer to an integer flag. If the socket's read pointer is currently at the out-of-band mark, the flag is set to a nonzero value. If it is not, the flag is set to zero.

SIOCDELRT Delete an entry from the interface routing table. Valid for sockets with address family of `AF_INET`.

See *SIOCADDRT* (page 70) for more information on the third parameter.

SIOCGIFADDR Get the interface address. Valid for sockets with address family of `AF_INET`.

The third parameter represents a pointer to the structure **ifreq**, defined in **<net/if.h>**:

```
struct ifreq {
   char ifr_name[IFNAMSIZE];
   union {
    struct sockaddr ifru_addr;
    struct sockaddr ifru_mask;
    struct sockaddr ifru_broadaddr;
    short ifru_flags;
    int ifru_mtu;
    int infu_rbufsize;
    char ifru_linename[10];
    char ifru_TOS;
   } ifr_ifru;
};
```

ifr_name is the name of the interface for which information is to be retrieved. The i5/OS implementation requires this field to be set to a NULL-terminated string that represents the interface IP address in dotted decimal format. Depending on the request, one of the fields in the *ifr_ifru* union will be set upon return from the *ioctl()* call. *ifru_addr* is the local IP address of the interface. *ifru_mask* is the subnetwork mask associated with the interface. *ifru_broadaddr* is the broadcast address. *ifru_flags* contains flags that give some information about an interface (for example, token-ring routing support, whether interface is active, broadcast address, and so on). *ifru_mtu* is the maximum transfer unit configured for the interface. *ifru_rbufsize* is the reassembly buffer size of the interface. *ifru_linename* is the line name associated with the interface. *ifru_TOS* is the type of service configured for the interface.

SIOCGIFBRDADDR Get the interface broadcast address. Valid for sockets with address family of `AF_INET`.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGIFCONF Get the interface configuration list. Valid for sockets with address family of AF_INET.

The third parameter represents a pointer to the structure **ifconf**, defined in **<net/if.h>**:

```
struct ifconf [
   int ifc_len;
   int ifc_configured;
   int ifc_returned;
   union {
    caddr_t ifcu_buf;
    struct ifreq *ifcu_req;
   } ifc_ifcu;
];
```

ifc_len is a value-result field. The caller passes the size of the buffer pointed to by *ifcu_buf*. On return, *ifc_len* contains the amount of storage that was used in the buffer pointed to by *ifcu_buf* for the interface entries. *ifc_configured* is the number of interface entries in the interface list. *ifc_returned* is the number of interface entries that were returned (this is dependent on the size of the buffer pointed to by *ifcu_buf*). *ifcu_buf* is the user buffer in which a list of interface entries will be stored. Each stored entry will be an *ifreq* structure.

To get the interface configuration list, the following fields must be set:

- *ifc_len*
- *ifcu_buf*

See SIOCGIFADDR (page 71) for more information on the list of *ifreq* structures returned. For this request, the *ifr_name* and *ifru_addr* fields will be set to a value.

Note: Additional information about each individual interface can be obtained using these values and the other interface-related requests.

SIOCGIFFLAGS Get interface flags. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGIFLIND Get the interface line description name. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGIFMTU Get the interface network MTU. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGIFNETMASK Get the mask for the network portion of the interface address. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGIFRBUFS Get the interface reassembly buffer size. Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGIFTOS Get the interface type-of-service (TOS). Valid for sockets with address family of AF_INET.

See *SIOCGIFADDR* (page 71) for more information on the third parameter.

SIOCGPGRP Get the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

See *FIOGETOWN* (page 69) for more information on the third parameter.

SIOCGRTCONF Get the route configuration list. Valid for sockets with address family of AF_INET.

For the SIOCGRTCONF request, the third parameter represents a pointer to the structure **rtconf**, also defined in **<net/route.h>**:

```
struct rtconf [
   int rtc_len;
   int rtc_configured;
   int rtc_returned;
   union {
    caddr_t rtcu_buf;
    struct rtentry *rtcu_req;
   } rtc_rtcu;
];
```

rtc_len is a value-result field. The caller passes the size of the buffer pointed to by *rtcu_buf*. On return, *rtc_len* contains the amount of storage that was used in the buffer pointed to by *rtcu_buf* for the route entries. *rtc_configured* is the number of route entries in the route list. *rtc_returned* is the number of route entries that were returned (this is dependent on the size of the buffer pointed to by *rtcu_buf*). *rtcu_buf* is the user buffer in which a list of route entries will be stored. Each stored entry will be an *rtentry* structure.

To get the route configuration list, the following fields must be set:

- *rtc_len*
- *rtcu_buf*

See SIOCADDRT (page 70) for more information on the list of *rtentry* structures returned. For this request, all fields in each *rtentry* structure will be set to a value.

SIOCSENDQ Return the number of bytes on the send queue that have not been acknowledged by the remote system. Valid for sockets with address family of AF_INET or AF_INET6 and socket type of SOCK_STREAM.

The third parameter represents a pointer to an integer that is set to the number of bytes yet to be acknowledged as being received by the remote TCP transport driver.

Notes:

1. SIOCSENDQ is used after a series of blocking or non-blocking send operations to see if the sent data has reached the transport layer on the remote system. Note that this does not not guarantee the data has reached the remote application.
2. When SIOCSENDQ is used in a multithreaded application, the actions of other threads must be considered by the application. SIOCSENDQ provides a result for a socket descriptor at the given point in time when the *ioctl()*) request is received by the TCP transport layer. Blocking send operations that have not completed, as well as non-blocking send operations in other threads issued after the SIOCSENDQ *ioctl()*, are not reflected in the result obtained for the SIOCSENDQ *ioctl()*.
3. In a situation where the application has multiple threads sending data on the same socket descriptor, the application should not assume that all data has been received by the remote side when 0 is returned if the application is not positive that all send operations in the other threads were complete at the time the SIOCSENDQ *ioctl()* was issued. An application should issue the SIOCSENDQ *ioctl()* only after it has completed all of the send operations. No value is added by querying the machine to see if it has sent all of the data when the application itself has not sent all of the data in a given unit of work.

SIOCSPGRP Set the process ID or process group ID that is to receive the SIGIO and SIGURG signals.

See *FIOSETOWN* (page 69) for more information on the third parameter.

Authorities

No authorization is required.

Return Value

0 **ioctl()** was successful

-1 **ioctl()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **ioctl()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.
[EAGAIN]	
[EBADF]	
[EBADFID]	
[EBUSY]	
[EDAMAGE]	
[EFAULT]	
[EINTR]	
[EINVAL]	
[EIO]	
[ENOBUFS]	
[ENOSPC]	
[ENOSYS]	
[ENOTAVAIL]	
[ENOTSAFE]	
[EPERM]	
[EPIPE]	
[ERESTART]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNATCH]	
[EUNKNOWN]	

If interaction with a file server is required to access the object, *errno* could also indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ETIMEDOUT]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPFA0D4 E	File system error occurred. Error number &1.
CPFA081 E	Unable to set return value or error code.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – ″Root″ (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400
2. QDLS File System Differences

   QDLS does not support **ioctl()**.
3. QOPT File System Differences

   QOPT does not support **ioctl()**.
4. A program must have the appropriate privilege `*IOSYSCFG` to issue any of the following requests: `SIOCADDRT` and `SIOCDELRT`.

Related Information

- The <**sys/ioctl.h**> file (see Header Files for UNIX-Type Functions)
- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- Socket Programming

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

listen()—Invite Incoming Connections Requests

Syntax

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

int listen(int socket_descriptor,
           int back_log)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *listen()* function is used to indicate a willingness to accept incoming connection requests. If a *listen()* is not done, incoming connections are silently discarded.

Parameters

socket_descriptor
(Input) The descriptor of the socket that is to be prepared to receive incoming connection requests.

back_log
(Input) The maximum number of connection requests that can be queued before the system starts rejecting incoming requests. The maximum number of connection requests that can be queued is defined by {SOMAXCONN} (defined in **<sys/socket.h>**).

Authorities

No authorization is required.

Return Value

listen() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *listen()* fails, *errno* can be set to one of the following:

[EADDRNOTAVAIL]	Address not available. The socket has an address family of `AF_INET` or `AF_INET6`, the socket was not bound, and the system tried to bind the socket but could not because a port was not available.
[EBADF]	Descriptor not valid.
[EINVAL]	Parameter not valid. This error code indicates one of the following: • A *connect()* has been issued on the socket pointed to by the *socket_descriptor* parameter. • The *socket_descriptor* parameter points to a socket with an address family of `AF_UNIX` that has not been bound to an address.
[EIO]	Input/output error.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOTSOCK]	The specified descriptor does not reference a socket.
[EOPNOTSUPP]	Operation not supported. The *socket_descriptor* parameter points to a socket that does not support *listen()*. *listen()* is only supported on sockets that are using a connection-oriented protocol (socket type of `SOCK_STREAM`).
[EUNKNOWN]	Unknown system state.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

Error Messages

CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. If the socket is not bound to an address and the address family is:
   - `AF_INET`, the system automatically selects an address (`INADDR_ANY` and an available port number) and binds it to the socket.
   - `AF_INET6`, the system automatically selects an address (`in6addr_any` and an available port number) and binds it to the socket.
   - `AF_UNIX`, the *listen()* fails with `[EINVAL]`.
2. *listen()* can be issued multiple times for a particular socket.
3. If the *back_log* parameter specifies a value greater than the maximum {SOMAXCONN} allowed, the specified value will be ignored and SOMAXCONN will be used. If the *back_log* parameter specifies a negative value, the specified value will be ignored and zero will be used.
4. The optimal setting of the *listen() back_log* value is dependent on the following factors:
   - The design of the server—how the server processes connection requests. Does it handle each connection request itself or does it pass the actual processing of the connection to a child or worker job? In other words, how long does it take for the server to handle an incoming connection until it can handle the next one? The shorter the time, the smaller the *back_log* value can be.
   - The number and rate of connection requests the server can expect over a given period of time will help determine the *back_log* value. More connection requests coming in over a shorter period of time requires a larger *back_log* value.
   - The following may determine how the server performs and thus how long it will take for an accept request to be serviced:
     – The system processor size
     – How storage pools used by the server are allocated
     – Machine performance

     The faster the server performance, the smaller the *back_log* value can be.

   Also, to help you determine how much main storage is consumed by a connection request in the *listen() back_log*, consider the following:
   - Each connection request in the backlog consumes at least 1KB of storage.
   - Each connection request can consume an additional storage amount equal to the size of TCP receive buffer. You can determine the TCP receive buffer size by looking at the TCPRCVBUF parameter value on the Change TCP Attributes (CHGTCPA) CL command. This storage amount will be consumed only if the remote peer (client) sends data after the connection is established and put into the backlog.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *listen()* API is mapped to *qso_listen98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

- "accept()—Wait for Connection Request and Make Connection" on page 4—Wait for Connection Request and Make Connection

API introduced: V3R1

---

poll()—Wait for Events on Multiple Descriptors

Syntax

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

int poll(struct pollfd fds[],
         nfds_t nfds,
         int timeout)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 79.

The *poll()* function is used to enable an application to multiplex I/O over a set of descriptors. For each member of the array pointed to by *fds, poll()* will examine the given descriptor for the event(s) specified. *nfds* is the number of **pollfd** structures in the *fds* array. The *poll()* function will determine which descriptors can read or write data, or whether certain events have occured on the descriptors within the *timeout* period.

Parameters

fds (Input) Specifies an array of descriptors to be examined and the events of interest for each descriptor.

```
struct pollfd {
   int    fd;         /* Descriptor        */
   short  events;     /* Requested events  */
   short  revents;    /* Returned events   */
};
```

The array's members are **pollfd** structures within which *fd* specifies an open descriptor. The *events* and *revents* are bitmasks constructed by OR'ing a combination of the following event flags.

The following events can be specified in the *events* field:

POLLIN	Data may be read without blocking.
POLLPRI	High-priority(OOB) data may be read without blocking.
POLLOUT	Data may be written without blocking.
POLLWRNORM	Equivalent to POLLOUT.
POLLRDNORM	Equivalent to POLLIN.
POLLNORM	Equivalent to POLLIN.

Each of the *events* listed above may be returned in the *revents* field when the *poll()* API completes if that requested condition is valid for the specified descriptor. In addition to these events, the following event may also be returned in the *revents* field:

POLLNVAL	The specified *fd* value is invalid. This flag is ignored if it is specified in the *events* field.

The following events will never be returned in the *revents* field on the iSeries:

POLLRDBAND	Priority message ready to read.
POLLWRBAND	Writable priority band exists.
POLLERR	An error occurred.
POLLHUP	Connection disconnected.

In each **pollfd** structure, *poll()* will clear the *revents* member, except that where the application requested a report on a condition by setting one of the bits of *events*, *poll()* will set the corresponding bit in *revents* if the requested condition is true. In addition, *poll()* shall set the POLLNVAL flag in *revents* if the condition is true, even if the application did not set the corresponding bit in events.

nfds (Input) *nfds* is the number of **pollfd** structures in the *fds* array.

timeout

(Input) The *timeout* argument specifies how long *poll()* is to wait before returning. If none of the requested events have occurred on any of the descriptors, *poll()* will wait at least *timeout* milliseconds for an event to occur on any of the descriptors. If the value of *timeout* is 0, *poll()* will return immediately. If the value of *timeout* is -1, *poll()* will block until a requested event occurs or until the call is interrupted.

Authorities

No authorization is required.

Return Value

poll() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (the *timeout* expired before any of the requested *events* were satisfied)
- n (total number of descriptors that met poll criteria)

Error Conditions

When *poll()* fails, *errno* can be set to one of the following:

[EINVAL]	The *timeout* argument was less than -1.
[ENOTSAFE]	Function not allowed.
[EINTR]	A signal was caught during the *poll()*.
[EFAULT]	The address used for an argument was not valid.
[EUNKNOWN]	Unknown system state.
[ENOTSUP]	Operation not supported.

Error Messages

CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. When specifying the POLLIN flag for *events*, the following can be indicated:
   - Data is available to be read
   - An error event exists on the descriptor.

- A socket descriptor that is listening for connections will indicate that it is ready for reading, once connections are available.
- No data can be read from the underlying instance represented by the descriptor. For example, a socket descriptor for which a *shutdown()* call has been done to disable the reception of data.

2. When specifying the POLLOUT flag for *events*, the following can be indicated:
   - When a *write()* can be successfully issued without blocking (or, for nonblocking, so it does not return [EWOULDBLOCK]).
   - Completion of a non-blocking *connect()* call on a socket descriptor. This allows an application to set a socket descriptor to nonblocking (with *fcntl()* or *ioctl()*), issue a *connect()* and receive [EINPROGRESS], and then use *poll()* to verify that the connection has completed.
   - No data can be written to the underlying instance represented by the descriptor (for example, a socket descriptor for which a *shutdown()* has been done to disable the sending of data).
3. If the *revents* field is set to POLLPRI when the *poll()* API completes, this indicates that out-of-band data has arrived on the descriptor specified by the *fd* field. This is only supported for connection-oriented sockets with an address family of AF_INET or AF_INET6.
4. The *poll()* API will not be affected by the O_NONBLOCK flag.
5. The *poll()* API is more efficient than the *select()* API and therefore *poll()* is always recommended over *select()*.
6. The timeout mechanism is different between *poll()* and *select()*. The *poll()* API uses an integer with the unit of measure as milliseconds. The *select()* API uses a timeval structure.
7. Unpredictable results will appear if this function or any of its associated type and macro definitions are used in a thread executing one of the scan-related exit programs (or any of its' created threads). See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.

Related Information

- "select()—Wait for Events on Multiple Sockets" on page 140—Wait for events on Multiple Sockets

API introduced: V5R4

Top | UNIX-Type APIs | APIs by category

QsoCancelOperation()—Cancel an I/O Operation

Syntax

```
#include <qsoasync.h>

int QsoCancelOperation(int socketDescriptor, unsigned long long operationId)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoCancelOperation() function is used to cancel one or more asynchronous I/O operations that are pending on the socket. Pending operations are defined as incomplete operations that have not been posted to an I/O completion port. The canceled operations will be posted to the I/O completion port with an *errnoValue* of *ECANCELED*.

If any operations that match the operation identifier are uninterruptible, then no pending operations will be cancelled. The only operation that is uninterruptible is gsk_secure_soc_startInit() when secure negotiations have already begun.

Parameters

int socketDescriptor (Input)
The socket descriptor where the operation was started.

unsigned long long operationId (Input)
The operation identifier that was specified in field *operationId* in the *Qso_OverlappedIO_t* structure when the operation was started.

All pending operations on the socket that match the *operationId* will be cancelled.

Authorities

No authorization is required.

Return Values

QsoCancelOperation() returns an integer. Possible values are:

- -1 - The function did not complete because an error occurred. Inspect the **errno** value to determine the cause of the failure.
- 0 - An operation matching the operation identifier was not pending and could not be cancelled.
- >0 - Successful, at least one operation was cancelled. The return value is the number of operations cancelled.

Errno Conditions

When QsoCancelOperation() fails, errno can be set to one of the following:

[EAGAIN]	At least one pending operation was uninterruptible. If there were multiple operations pending, none were cancelled.
[EBADF]	Invalid descriptor.
[ENOTSOCK]	The specified descriptor is not a socket.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. When *QsoCancelOperation()* fails with EAGAIN for multiple pending operation, the application may retry *QsoCancelOperation()* and some pending operations may yet be cancelled after the uninterruptible operation has completed. Or the application may *close()* the socket to force all pending operations to be cancelled.

Related Information

- "QsoCreateIOCompletionPort()—Create I/O Completion Port"—Create I/O Completion Port
- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port" on page 83—Destroy I/O Completion Port
- gsk_secure_soc_startInit()—Start Asynchronous Operation to negotiate a secure session
- gsk_secure_soc_startRecv—Start Asynchronous Recv Operation on a secure session
- gsk_secure_soc_startSend—Start Asynchronous Send Operation on a secure session
- "QsoGenerateOperationId()—Get an I/O Operation ID" on page 85—Get an I/O Operation ID
- "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86—Check if an I/O Operation is Pending
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation

API introduced: V5R4 with PTF

QsoCreateIOCompletionPort()—Create I/O Completion Port

Syntax

```
#include <qsoasync.h>
int QsoCreateIOCompletionPort()
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoCreateIOCompletionPort is used to create a common wait point for a completed overlapped I/O operation. The wait point is represented by the I/O completion port handle returned by the QsoCreateIOCompletionPort() function. This handle is specified on QsoStartRecv and QsoStartSend functions to initiate overlapped I/O operations.

Authorities

No authorization is required.

Return Values

QsoCreateIOCompletionPort() returns an integer. Possible values are:

- -1 - Unsuccessful, errno is set to a value defined below.
- n - Successful, where n is an I/O completion port handle that can be used in conjunction with overlapped I/O functions QsoStartRecv(), QsoStartSend(), and QsoPostIOCompletionPort().

Errno Conditions

When QsoCreateIOCompletionPort() fails, errno can be set to one of the following:

[ENOBUFS] The limit of 256 I/O completion ports has been exceeded for this process.
[EUNKNOWN] Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The I/O completion port handle is a process scoped resource; therefore, you may not start an overlapped I/O function on a socket in one process and check for its completion in another process.
2. The number of I/O completion ports that can be active for a given process is 256.

Related Information

- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port"—Create I/O Completion Port
- "QsoPostIOCompletion()—Post I/O Completion Request" on page 87—Post Request on I/O Completion Port
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QsoDestroyIOCompletionPort()—Destroy I/O Completion Port

Syntax

```
#include <qsoasync.h>

int QsoDestroyIOCompletionPort
(int IOCompletionPort)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoDestroyIOCompletionPort is used to destroy an I/O completion port.

Parameters

int IOCompletionPort (Input)

The I/O completion port to be destroyed. All threads sleeping with *QsoWaitForIOCompletion()* on the I/O completion port being destroyed will be awakened with return value of -1 and errno value of EDESTROYED.

Authorities

No authorization is required.

Return Values

QsoDestroyIOCompletionPort() returns an integer. Possible values are:

- 0 - Successful destruction of the I/O completion port.
- -1 - The function has failed. Inspect the errno value to determine the cause of the failure.

Errno Conditions

When QsoDestroyIOCompletionPort fails, errno can be set to one of the following:

[EINVAL]	The specified I/O completion port is not valid.
[EUNKOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. There can be many overlapped I/O operations outstanding when an I/O completion port is destroyed. The buffers that are associated with these overlapped I/O operations are available for use by the application as soon as *QsoDestroyIOCompletionPort()*returns successfully.
2. The state of the sockets that were used to issue the overlapped I/O operations that are still outstanding is not defined. That is, there is no way for the application to determine if an outstanding *QsoStartRecv()* or *QsoStartSend()* has completed once the I/O completion port has been destroyed. For this reason, further attempts to read from those sockets will result in ECONNABORTED and further attempts to write to these sockets will result in EPIPE. No further input or output operations will be allowed on these sockets.

Related Information

- "QsoCreateIOCompletionPort()—Create I/O Completion Port" on page 82—Create I/O Completion Port
- "QsoPostIOCompletion()—Post I/O Completion Request" on page 87—Post Request on I/O Completion Port
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QsoGenerateOperationId()—Get an I/O Operation ID

Syntax

```
#include <qsoasync.h>

unsigned long long QsoGenerateOperationId(int socketDescriptor)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoGenerateOperationId() function is used to get an operation identifier that is unique for this socket. The operation identifier may then be used in field *operationId* in the *Qso_OverlappedIO_t* structure when an asynchronous I/O operation is started.

It is not required that an application use QsoGetIoID() to set the I/O identifer. Any appropriate application defined value may be used. Individual operations may use unique operation identifiers or groups of operations could share I/O identifiers, depending on the application's requirements. QsoGenerateOperationId(), when used consistently, is a convenient means to get unique identifiers for use on a socket. Note that operation identifiers from one sockets may not be unique if used on a different socket.

I/o identifiers are ignored by all API's except "QsoCancelOperation()—Cancel an I/O Operation" on page 80 and "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86. Other start operations will only preserve the input value and return it on "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100.

Parameters

int socketDescriptor (Input)
The socket descriptor where the operation identifier will be used.

Authorities

No authorization is required.

Return Values

QsoGenerateOperationId() returns an unsigned long long operation identifier. Possible values are:

- 0 - The function did not complete because an error occurred. Inspect the **errno** value to determine the cause of the failure.
- <>0 - Successful, the value returned is a unique operation identifier for the socket.

Errno Conditions

When QsoGenerateOperationId() fails, errno can be set to one of the following:

[EBADF]	Invalid descriptor
[ENOTSOCK]	The specified descriptor is not a socket.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Related Information

- gsk_secure_soc_startInit()—Start Asynchronous Operation to negotiate a secure session
- gsk_secure_soc_startRecv—Start Asynchronous Recv Operation on a secure session
- gsk_secure_soc_startSend—Start Asynchronous Send Operation on a secure session
- "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation
- "QsoIsOperationPending()—Check if an I/O Operation is Pending"—Check if an I/O Operation is Pending
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation

«

API introduced: V5R4 with PTF

Top | UNIX-Type APIs | APIs by category

QsoIsOperationPending()—Check if an I/O Operation is Pending

Syntax

```
#include <qsoasync.h>

int QsoIsOperationPending(int socketDescriptor, unsigned long long operationId)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoIsOperationPending() function is used to check if one or more asynchronous I/O operations is pending on the socket. Pending operations are defined as incomplete operations that have not been posted to an I/O completion port.

Parameters

int socketDescriptor (Input)
The socket descriptor from which to generate an operation identifier.

unsigned long long operationId (Input)
The operation identifier that was specified in field *operationId* in the *Qso_OverlappedIO_t* structure when the operation was started.

Authorities

No authorization is required.

Return Values

QsoIsOperationPending() returns an integer. Possible values are:

- -1 - The function did not complete because an error occurred. Inspect the **errno** value to determine the cause of the failure.
- 0 - An operation matching the operation identifier was not pending.
- >0 - Successful, at least one operation that matched the operation identifier was pending. The return value is the number of matching operations currently pending.

Errno Conditions

When QsoIsOperationPending() fails, errno can be set to one of the following:

[EBADF]	Invalid descriptor
[ENOTSOCK]	The specified descriptor is not a socket.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Related Information

- gsk_secure_soc_startInit()—Start Asynchronous Operation to negotiate a secure session
- gsk_secure_soc_startRecv—Start Asynchronous Recv Operation on a secure session
- gsk_secure_soc_startSend—Start Asynchronous Send Operation on a secure session
- "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation
- "QsoGenerateOperationId()—Get an I/O Operation ID" on page 85—Get an I/O Operation ID
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation

API introduced: V5R4 with PTF

Top | UNIX-Type APIs | APIs by category

QsoPostIOCompletion()—Post I/O Completion Request

Syntax

```
#include <qsoasync.h>

int QsoPostIOCompletion
(int IOCompletionPort, Qso_OverlappedIO_t * communicationsArea)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoPostIOCompletion function will post an Qso_OverlappedIO_t request on a specifed I/O completion port. This allows an application to notify a completion port that some function or activity has occurred. The application defines what that function or activity is within the Qso_OverlappedIO_t request.

Parameters

int IOCompletionPort (Input)
The I/O completion port that should be posted.

Qso_OverlappedIO_t * communicationsArea (Input/Output)
A pointer to a structure that contains the following information:

descriptorHandle	(Input) - The descriptor handle is application-specific and is never used by the system. It is intended to make it easier for the application to keep track of information regarding a given socket connection.
buffer	(Input) - Supplied value is preserved.
bufferLength	(Input) - Supplied value is preserved.
postFlag	(Input) - Supplied value is preserved.
fillBuffer	(Input) - Supplied value is preserved.
returnValue	(Output) - This field will be set to 0 if this operation completes successfully.
errnoValue	(Output) - This field will be set to 0 if this operation completes successfully.
operationCompleted	(Output) - When the operation is posted to the I/O completion port, this field is updated to indicate that the operation was a QSOPOSTIOCOMPLETION.
secureDataTransferSize	Not used.
bytesAvailable	Not used.

operationWaitTime (Input) - A timeval structure which specifies a time to wait before posting this operation asynchronously to the I/O completion port with *errnoValue* set to EAGAIN.

```
struct timeval {
                long  tv_sec;   /* second          */
                long  tv_usec;  /* microseconds    */
};
```

If this field is set to zero, the operation will be posted immediately.

If this field is non-zero, then the *postedDescriptor* field must be set.

If *postedDescriptor* is closed before the timer expires, the operation will be posted to the I/O completion port with *errnoValue* set to ECLOSED.

The minimum operationWaitTime is 1 second. The microseconds field (tv_usec) in the timeval is not used and must be set to zero.

postedDescriptor This field is only relevant if a non-zero timeval was specified in *operationWaitTime*. This is the socket descriptor to be associated with the timer. If this descriptor is closed before the timer expires, the operation will be posted to the I/O completion port with *errnoValue* set to ECLOSED.

This field must be set when the *operationWaitTime* field is used.

operationId (Input) - An identifier to uniquely identify this operation or a group of operations. It can be set with the return value from "QsoGenerateOperationId()—Get an I/O Operation ID" on page 85 or with an application-defined value.
This value is preserved but ignored by all APIs except "QsoCancelOperation()—Cancel an I/O Operation" on page 80 and "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86. This field is applicable only when both *postedDescriptor* and *operationWaitTime* are specified. Otherwise the operation completes immediately and cannot be cancelled.

reserved1 (Input) - Must be set to hex zeroes.

reserved2 (Input) - Must be set to hex zeroes.

Authorities

No authorization is required.

Return Values

QsoPostIOCompletion() returns an integer. Possible values are:

- -1 - The function did not complete because an error occurred. Inspect the **errno** value to determine the cause of the failure.
- 0 - The function has successfully posted the communications area to the I/O completion port.
- 1 - The timer has been started. When the timer expires the Qso_OverlappedIO_t communications structure will be updated with the results and the I/O completion port will be posted.

Errno Conditions

When QsoPostIOCompletion() fails, errno can be set to one of the following:

[EINVAL]	The I/O completion port or a reserved field was specified that was not valid or operationWaitTime.tv_sec was negative or operationWaitTime.tv_usec was not zero.
[EDESTROYED]	The I/O completion port has been destroyed.
[ENOTSOCK]	The *postedDescriptor* field was not set to a socket descriptor when *operationWaitTime* was set.
[ENOBUFS]	There was not enough buffer space for the requested operation. Check the maximum allowed storage for the executing user profile.
[ENOMEM]	The I/O completion port is full and cannot accept any more messages at this time.

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Related Information

- "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation
- "QsoCreateIOCompletionPort()—Create I/O Completion Port" on page 82—Create I/O Completion Port
- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port" on page 83—Destroy I/O Completion Port
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

QsoStartAccept()—Start asynchronous accept operation

Syntax

```
#include <sys/socket.h>
#include <qsoasync.h>

int QsoStartAccept (int socketDescriptor,int IOCompletionPort,
Qso_OverlappedIO_t * communicationsArea)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The **QsoStartAccept()** function is used to wait asynchronously for connection requests. If connection requests are queued, then QsoStartAccept() takes the first connection request on the queue and creates a new socket to service the connection request. If no connection requests are queued, then an asynchronous QsoStartAccept() request is pended onto the socket and will be transition to the specified I/O completion port once a connection arrives. This API only supports sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

Parameters

socketDescriptor (Input)
The descriptor of the socket on which to wait.

int IOCompletionPort(Input)
The I/O completion port that should be posted when the operation completes.

Qso_OverlappedIO_t* communicationsArea (Input/Output)
A pointer to a structure that contains the following information:

descriptorHandle	(Input) - The descriptor handle is application specific and is never used by the system. This field is intended to make it easier for the application to keep track of information regarding a given socket connection.
buffer	Not used.
bufferLength	Not used.
postFlag	(Input) - The postFlag indicates if this operation should be posted to the I/O completion port even if it completes immediately. • A 0 value indicates that if the operation is already complete upon return to the application, then do not post to the I/O completion port. • A 1 value indicates that even if the operation completes immediately upon return to the application, the result should still be posted to the I/O completion port.
postFlagResult	» Not used. «
fillBuffer	Not used.
returnValue	When QsoStartAccept() completes synchronously (function return value equals 0), then this field identifies the socket descriptor associated with the accepted connection. When the accept operation completes asynchronously, this field contains indication of success or failure.
errnoValue	(Output) - When the operation completes asynchronously and returnValue is negative, this field will contain an errno to indicate the error with which the operation eventually failed.
operationCompleted	(Output) - If the operation is posted to the I/O completion port, this field is updated to indicate that the operation was a QSOSTARTACCEPT.
secureDataTransferSize	Not used.
bytesAvailable	(Output) - Number of bytes available to be read from connection. Only valid if returnValue is >=0.

operationWaitTime (Input) - A timeval structure which specifies the maximum time allowed for this operation to complete asynchronously.

```
struct timeval {
                long  tv_sec;   /* second          */
                long  tv_usec;  /* microseconds    */
};
```

If this timer expires, the operation will be posted to the I/O completion port with *errnoValue* set to EAGAIN.

If this field is set to zero, the operation's asynchronous completion will not be timed.

If socketDescriptor is closed before the operation completes or times out, the operation will be posted to the I/O completion port with *errnoValue* set to ECLOSED.

The minimum operationWaitTime is 1 second. The microseconds field (tv_usec) in the timeval is not used and must be set to zero.

postedDescriptor Not used - Must be set to zero.

» *operationId* (Input) - An identifier to uniquely identify this operation or a group of operations. It can be set with the return value from "QsoGenerateOperationId()—Get an I/O Operation ID" on page 85 or with an application-defined value.
This value is preserved but ignored by all APIs except "QsoCancelOperation()—Cancel an I/O Operation" on page 80 and "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86. «

reserved1 (Output) - Must be set to hex zeroes.

reserved2 (Input) - Must be set to hex zeroes.

Authorities

No authorization is required.

Return Values

QsoStartAccept() returns an integer. Possible values are:

- -1 - The function was not started because an error occurred. Inspect the **errno** to determine the cause of the failure.
- 0 - The function has already completed. The Qso_OverlappedIO_t communications structure has been updated but nothing has or will be posted to the I/O completion port for this operation. Inspect the returnValue in the Qso_OverlappedIO_t communications structure to obtain connection descriptor and bytesAvailable.
- 1 - The function has been started. When the function completes (or times out if operationWaitTime was specified), the Qso_OverlappedIO_t communications structure will be updated with the results and the I/O completion port will be posted.

Errno Conditions

When **QsoStartAccept()** fails, errno can be set to one of the following:

[EFAULT] Bad address

[EINVAL] A I/O completion port or reserved field specified was not valid or postedDescriptor was not zero or operationWaitTime.tv_sec was negative or operationWaitTime.tv_usec was not zero, or a Listen() has not been issued against the socket referenced by the SocketDescriptor parameter.

[EACCES] Permission denied.

A connection indication request was received on the socket referenced by the *socket_descriptor* parameter, but the process that issued the **QsoStartAccept()** did not have the appropriate privileges required to handle the request. The connection indication request is reset by the system.

[EBADF]	Descriptor not valid.
[ECONNABORTED]	Connection ended abnormally.
	An **QsoStartAccept()** was issued on a socket for which receives have been disallowed (due to a *shutdown()* call).
[EIO]	Input/output.
[EMFILE]	Too many descriptors for this process.
[ENFILE]	Too many descriptors in system.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOTSOCK]	The specified descriptor does not reference a socket.
[EOPNOTSUPP]	Operation not supported.
	The *socket_descriptor* parameter references a socket that does not support the **QsoStartAccept()**. The **QsoStartAccept()** is only valid on sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
	The *socket_descriptor* parameter references a socket that has undergone an Rbind(). The **QsoStartAccept()** operation is not valid on sockets in this state.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. It is not recommended to intermix QsoStartAccept() and accept(). If this condition occurs, the order the requests will be serviced is undefined.
2. The following are inherited by the descriptor returned by the accept() call:
   - All socket options with a level of SOL_SOCKET.
   - The status flags:
     - Blocking flag (set/reset either by the *iocltl()* call with the FIONBIO request or by the *fcntl()* call with the F_SETFL command and the status flag set to O_NONBLOCK).
     - Asynchronous flag (set/reset either by the *ioctl()* call with the FIOASYNC request or by the *fcntl()* call with the F_SETFL command and the status flag set to FASYNC).
   - The process ID or process group ID that is to receive SIGIO or SIGURG signals (set/reset by either the *ioctl()* call with the FIOSETOWN or the SIOCSPGRP request, or by the *fcntl()* call with the F_SETOWN command).
3. Closing a socket causes any queued but unaccepted connection requests to be reset.

Related Information

- "accept()—Wait for Connection Request and Make Connection" on page 4—Accept Connection
- "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation

- "QsoCreateIOCompletionPort()—Create I/O Completion Port" on page 82—Create I/O Completion Port
- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port" on page 83—Destroy I/O Completion Port
- "QsoPostIOCompletion()—Post I/O Completion Request" on page 87—Post Request on I/O Completion Port
- "QsoStartRecv()—Start Asynchronous Receive Operation"—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation
- "recv()—Receive Data" on page 119—Receive Data

API introduced: V5R1

QsoStartRecv()—Start Asynchronous Receive Operation

Syntax

```
#include <qsoasync.h>

int QsoStartRecv (int socketDescriptor,int IOCompletionPort,
Qso_OverlappedIO_t * communicationsArea)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoStartRecv function is used to initiate an asynchronous receive operation. The supplied buffer cannot be reused by the calling application until the receive is complete or the I/O completion port specified on the QsoStartRecv has been destroyed. This API only supports sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

Parameters

int socketDescriptor (Input)
The socket descriptor that should be used to receive data into the specified buffer.

int IOCompletionPort (Input)
The I/O completion port that should be posted when the operation completes.

Qso_OverlappedIO_t * communicationsArea (Input/Output)
A pointer to a structure that contains the following information:

descriptorHandle	(Input) - The descriptor handle is application specific and is never used by the system. This field is intended to make it easier for the application to keep track of information regarding a given socket connection.
buffer	(Input) - A pointer to a buffer into which data should be read.
bufferLength	(Input) - The length of the buffer into which data should be read. Also represents the amount of data requested.

postFlag (Input) - The postFlag indicates if this operation should be posted to the I/O completion port even if it completes immediately.

- A 0 value indicates that if the operation is already complete upon return to the application, then do not post to the I/O completion port.
- A 1 value indicates that even if the operation completes immediately upon return to the application, the result should still be posted to the I/O completion port.

postFlagResult (Output) - This field is valid if QsoStartRecv() returns with 1 and postFlag was set to 1. In this scenario, postFlagResult set to 1 denotes the operation completed and been posted to the I/O completion port specified. A value of 0 denotes the operation could not be completed immediately, but will be handled asynchronously.

fillBuffer (Input) - The fillBuffer flag indicates when this operation should complete. If the fillBuffer flag is 0, then the operation will complete as soon as any data is available to be received. If the fillBuffer flag is non-zero, this operation will not complete until enough data has been received to fill the buffer, an end-of-file condition occurs on the socket, or an error occurs on a socket.

returnValue (Output) - When QsoStartRecv() completes synchronously (function return value equals 0), then this field indicates the number of bytes that were actually received. When the recv operation completes asynchronously, this field contains indication of success or failure. Zero returned denotes end-of-file state.

errnoValue (Output) - When the operation completes asynchronously and returnValue is negative, this field contains an errno to indicate the error with which the operation eventually failed.

operationCompleted (Output) - If the operation is posted to the I/O completion port, this field is updated to indicate that the operation was a QsoStartRecv().

secureDataTransferSize Not used.

bytesAvailable Not used.

operationWaitTime (Input) - A timeval structure which specifies the maximum time allowed for this operation to complete asynchronously.

```
struct timeval {
                long  tv_sec;   /* second          */
                long  tv_usec;  /* microseconds    */
};
```

If this timer expires, the operation will be posted to the I/O completion port with *errnoValue* set to EAGAIN.

If this field is set to zero, the operation's asynchronous completion will not be timed.

If socketDescriptor is closed before the operation completes or times out, the operation will be posted to the I/O completion port with *errnoValue* set to ECLOSED.

The minimum operationWaitTime is 1 second. The microseconds field (tv_usec) in the timeval is not used and must be set to zero.

postedDescriptor Not used - Must be set to zero.

operationId (Input) - An identifier to uniquely identify this operation or a group of operations. It can be set with the return value from "QsoGenerateOperationId()—Get an I/O Operation ID" on page 85 or with an application-defined value.
This value is preserved but ignored by all APIs except "QsoCancelOperation()—Cancel an I/O Operation" on page 80 and "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86.

reserved1 (Input) - Must be set to hex zeroes.

reserved2 (Input) - Must be set to hex zeroes.

Authorities

No authorization is required.

Return Values

QsoStartRecv() returns an integer. Possible values are:

- -1 - The function was not started because an error occurred. Inspect the **errno** to determine the cause of the failure.
- 0 - The function has already completed. The Qso_OverlappedIO_t communications structure has been updated but nothing has or will be posted to the I/O completion port for this operation. Inspect the returnValue in the Qso_OverlappedIO_t communications structure to determine the number of bytes received.
- 1 - The function has been started. When the function completes (or times out if operationWaitTime was specified), the Qso_OverlappedIO_t communications structure will be updated with the results and the I/O completion port will be posted.

Errno Conditions

When QsoStartRecv() fails, errno can be set to one of the following:

[EINVAL]	A buffer length or I/O completion port or reserved field specified was not valid or postedDescriptor was not zero or operationWaitTime.tv_sec was negative or operationWaitTime.tv_usec was not zero.
[ETRUNC]	Data was truncated on an input, output, or update operation. Data has been lost.

Note: The rest of the errno values from "recv()—Receive Data" on page 119 also apply to QsoStartRecv().

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. If QsoStartRecv() partially fills a buffer and then encounters an EFAULT condition, the QsoStartRecv() will complete with the ETRUNC error value to indicate that some data has been lost.
2. A buffer that is given to QsoStartRecv() must not be used by the application again until either it is returned by QsoWaitForIOCompletion() or is reclaimed by issuing a close() on the socket descriptor or issuing a QsoDestroyIOCompletionPort() on the I/O completion port. If a buffer is given to QsoStartRecv() to be filled, and it is later detected during QsoStartRecv processing that the buffer has been freed, it may produce an unrecoverable condition on the socket for which the QsoStartRecv() was issued. If this occurs, an ECONNABORTED error value will be returned.
3. It is not recommended to intermix QsoStartRecv() and blocking I/O (that is, recv()) on the same socket. If this condition occurs, then pending asynchronous send I/O will be serviced first before the blocking I/O.
4. Socket option SO_RCVLOWAT is not supported by this API. Semantics similar to SO_RCVLOWAT can be obtained using the fillBuffer field in the Qso_OverLappedIO_t structure.
5. Socket option SO_RCVTIMEO is not supported by this API. Semantics similar to SO_RCVTIMEO can be obtained using the operationWaitTime field in the Qso_OverLappedIO_t structure.

Related Information

- » "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation

«
- "QsoCreateIOCompletionPort()—Create I/O Completion Port" on page 82—Create I/O Completion Port

- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port" on page 83—Create I/O Completion Port
- "QsoPostIOCompletion()—Post I/O Completion Request" on page 87—Post Request on I/O Completion Port
- "QsoStartSend()—Start Asynchronous Send Operation"—Start Asynchronous Send Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation
- "recv()—Receive Data" on page 119—Receive Data

API introduced: V5R1

QsoStartSend()—Start Asynchronous Send Operation

Syntax

```
#include <qsoasync.h>

int QsoStartSend (int socketDescriptor, int IOCompletionPort,
Qso_OverlappedIO_t * communicationsArea)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoStartSend function is used to initiate a asynchronous send operation. The supplied buffer cannot be reused by the calling application until the send is complete or the I/O completion port specified on the QsoStartSend has been destroyed. This API only supports sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

Parameters

int socketDescriptor (Input)
The socket descriptor on which the data should be sent.

int IOCompletionPort(Input)
The I/O completion port that should be posted when the operation completes.

Qso_OverlappedIO_t * communicationsArea (Input/Output)
A pointer to a structure that contains the following information:

descriptorHandle	(Input) - The descriptor handle is application specific and is never used by the system. This field is intended to make it easier for the application to keep track of information regarding a given socket connection.
buffer	(Input) - A pointer to a buffer of data that should be sent over the socket.
bufferLength	(Input) - The length of the data to be sent.
postFlag	(Input) - The postFlag indicates if this operation should be posted to the I/O completion port even if it completes immediately. • A 0 value indicates that if the operation is already complete upon return to the application, then do not post to the I/O completion port. • A 1 value indicates that even if the operation completes immediately upon return to the application, the result should still be posted to the I/O completion port.

postFlagResult (Output) - This field is valid if QsoStartSend() returns with 1 and postFlag was set to 1. In this scenario, postFlagResult set to 1 denotes the operation completed and been posted to the I/O completion port specified. A value of 0 denotes the operation could not be completed immediately, but will be handled asynchronously.

fillBuffer (Input) - Only used on QsoStartRecv(). Ignored on QsoStartSend().

returnValue (Output) - When QsoStartSend() completes synchronously (function return value equals 0), then this field indicates the number of bytes that was actually sent. When the send operation completes asynchronously, this filed contains indication of success or failure.

errnoValue (Output) - When the operation completes asynchronously and returnValue is negative, this field will contain an errno to indicate the error with which the operation eventually failed.

operationCompleted (Output) - If the operation is posted to the I/O completion port, this field is updated to indicate that the operation was a QsoStartSend().

secureDataTransferSize Not used.

bytesAvailable Not used.

operationWaitTime (Input) - A timeval structure which specifies the maximum time allowed for this operation to complete asynchronously.

```
struct timeval {
                long  tv_sec;   /* second          */
                long  tv_usec;  /* microseconds    */
};
```

If this timer expires, the operation will be posted to the I/O completion port with *errnoValue* set to EAGAIN.

If this field is set to zero, the operation's asynchronous completion will not be timed.

If socketDescriptor is closed before the operation completes or times out, the operation will be posted to the I/O completion port with *errnoValue* set to ECLOSED.

The minimum operationWaitTime is 1 second. The microseconds field (tv_usec) in the timeval is not used and must be set to zero.

postedDescriptor Not used - Must be set to zero.

» *operationId* (Input) - An identifier to uniquely identify this operation or a group of operations. It can be set with the return value from "QsoGenerateOperationId()—Get an I/O Operation ID" on page 85 or with an application-defined value.
This value is preserved but ignored by all APIs except "QsoCancelOperation()—Cancel an I/O Operation" on page 80 and "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86. «

reserved1 (Input) - Must be set to hex zeroes.

reserved2 (Input) - Must be set to hex zeroes.

Authorities

No authorization is required.

Return Values

QsoStartSend() returns an integer. Possible values are:

- -1 - The function was not started because an error occurred. Inspect the **errno** to determine the cause of the failure.
- 0 - The function has already completed. The Qso_OverlappedIO_t communications structure has been updated but nothing has or will be posted to the I/O completion port for this operation. Inspect the returnValue in the Qso_OverlappedIO_t communications structure to determine the number of bytes sent.
- 1 - The function has been started. When the function completes (or times out if operationWaitTime was specified), the Qso_OverlappedIO_t communications structure will be updated with the results and the I/O completion port will be posted.

Errno Conditions

When QsoStartSend() fails, errno can be set to one of the following:

[EINVAL] A buffer length or I/O completion port or reserved field specified was not valid or postedDescriptor was not zero or operationWaitTime.tv_sec was negative or operationWaitTime.tv_usec was not zero.

Note: The rest of the errno values from "send()—Send Data" on page 143 also apply to QsoStartSend().

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. It is important for application programmers to keep in mind that since QsoStartSend() is asynchronous, care should be used to control how many of these functions are outstanding. When a TCP socket becomes flow control blocked such that the QsoStartSend() is not able to pass the data to the TCP socket immediately, the return value will be 1. Applications that send large amounts of data should have the postFlag set to 0. This allows the application to use a return value of 1 as an indication that the socket has become flow control blocked. The application should then wait for the outstanding operation to complete before issuing another QsoStartSend(). This will ensure that the application does not exhaust system buffer resources.
2. A buffer that is given to QsoStartSend() must not be used by the application again until either it is returned by QsoWaitForIOCompletion() or is reclaimed by issuing a close() on the socket descriptor or issuing a QsoDestroyIOCompletionPort() on the I/O completion port. If a buffer is given to QsoStartSend() to be sent, and it is later detected during QsoStartSend() processing that the buffer has been freed, it may produce an unrecoverable condition on the socket for which the QsoStartSend() was issued. If this occurs, an ECONNABORTED error value will be returned.
3. It is not recommended to intermix QsoStartSend() and blocking I/O (that is, send()) on the same socket. If one does, then the pending asynchronous send I/O will be serviced before blocking I/O once data can be sent.
4. Socket option SO_SNDTIMEO is not supported by this API. Semantics similar to SO_SNDTIMEO can be obtained using the operationWaitTime field in the Qso_OverLappedIO_t structure.

Related Information

- » "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation «
- "QsoCreateIOCompletionPort()—Create I/O Completion Port" on page 82—Create I/O Completion Port
- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port" on page 83—Destroy I/O Completion Port
- "QsoPostIOCompletion()—Post I/O Completion Request" on page 87—Post Request on I/O Completion Port
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoWaitForIOCompletion()—Wait for I/O Operation" on page 100—Wait for I/O Completion Operation
- "send()—Send Data" on page 143—Send Data

API introduced: V5R1

QsoWaitForIOCompletion()—Wait for I/O Operation

Syntax

```
#include <gskssl.h>
#include <qsoasync.h

int QsoWaitForIOCompletion (int IOCompletionPort,
Qso_OverlappedIO_t * completionStatus,
struct timeval * timeToWait)
```

Service Program Name: QSOSRV3
Default Public Authority: *USE
Threadsafe: Yes

The QsoWaitForIOCompletion() is used to wait for a completed overlapped I/O operation. The wait point is represented by the I/O completion port that was created using the *QsoCreateIOCompletionPort()* function.

Parameters

int IOCompletionPort
(Input) The I/O completion port on which to wait.

Qso_OverlappedIO_t * completionStatus
(Input/Output) A pointer to a qso_overlappedIO_t structure that will be updated with the status defined below. If a field has no relevance to operation completed, then either a null or zero will be returned for that field.

descriptorHandle	(Output) The descriptor handle that was supplied by the application when the operation was started.
buffer	(Output) A pointer to the buffer that was supplied when the operation was started. Null is returned when operationCompleted is QSOSTARTACCEPT or GSKSECURESOCSTARTINIT .
bufferLength	(Output) The length of the buffer that was supplied when the operation was started. Zero is returned when operationCompleted is QSOSTARTACCEPT or GSKSECURESOCSTARTINIT.
postFlag	(Output) The value of the postFlag when the operation was started. Zero is returned when operationCompleted is QSOSTARTACCEPT or GSKSECURESOCSTARTINIT .
fillBuffer	(Output) The value of the fillBuffer when the operation was started. Zero is returned when operationCompleted is QSOSTARTACCEPT or GSKSECURESOCSTARTINIT .

returnValue (Output)

Possible values if operation completed is QSOPOSTIOCOMPLETION, QSOSTARTRECV, QSOSTARTSEND, or QSOSTARTACCEPT:

-1 The operation failed and errnoValue field should be checked for further explanation of the error.

>= 0 For both QSOSTARTRECV and QSOSTARTSEND, indicates the number of bytes sent or received respectively. A return value of 0 on a receive indicates an end-of-file condition. For QSOSTARTACCEPT, this field is the socket connection descriptor. For QSOPOSTIOCOMPLETION, a return value of 0 indicates the operation was not timed (*operationWaitTime* was zero on input). QSOPOSTIOCOMPLETION will not return > 0.

Possible values if operation completed is GSKSECURESOCSTARTSEND or GSKSECURESOCSTARTRECV:

GSK_OK
Operation was successful. Field secureDataTransferSize indicates the number of bytes sent or received respectively.

Failure

Possible values common to GSKSECURESOCSTARTSEND and GSKSECURESOCSTARTRECV:

[GSK_AS400_ERROR_INVALID_POINTER]
The buffer pointer located in the Qso_OverLappedIO_t is not valid.

[GSK_INTERNAL_ERROR]
An unexpected error occurred during SSL processing.

[GSK_AS400_ERROR_CLOSED]
Secure session was closed by a thread during SSL processing.

[GSK_ERROR_IO]
An error occurred in SSL processing; check the errno value.

[GSK_ERROR_SOCKET_CLOSED]
A **close()** was done on the socket descriptor for this secure session.

Values unique to GSKSECURESOCSTARTRECV:

[GSK_INVALID_HANDLE]
The handle specified was not valid.

[GSK_INVALID_STATE]
The handle is not in the correct state for this operation.

[GSK_ERROR_BAD_MESSAGE]
SSL received a badly formatted message.

[GSK_ERROR_BAD_MAC]
A bad message authentication code was received.

Possible values if operationCompleted is GSKSECURESOCSTARTINIT:

[GSK_OK]
Operation was successful, a secure session established.

[GSK_ERROR_BAD_MESSAGE]
SSL received a badly formatted message.

[GSK_ERROR_BAD_MAC]
A bad message authentication code was received.

[GSK_KEYRING_OPEN_ERROR]
Certificate store file could not be opened.

[GSK_ERROR_BAD_KEYFILE_LABEL]
The specified certificate store label is not valid.

[GSK_ERROR_BAD_V3_CIPHER]
An SSLV3 or TLSV1 cipher suite was specified that is not valid.

[GSK_ERROR_BAD_V2_CIPHER]
An SSLV2 cipher suite was specified that is not valid.

[GSK_ERROR_NO_CIPHERS]
No ciphers available or no ciphers were specified.

[GSK_ERROR_NO_CERTIFICATE]
No certificate is available for SSL processing.

[GSK_ERROR_BAD_CERTIFICATE]
The certificate is bad.

[SSL_ERROR_NOT_TRUSTED_ROOT]
The certificate is not signed by a trusted certificate authority.

[GSK_KEYFILE_CERT_EXPIRED]
The validity time period of the certificate has expired.

[GSK_ERROR_BAD_MESSAGE]
A badly formatted message was received.

[GSK_ERROR_UNSUPPORTED]
Operation is not supported by SSL.

[GSK_ERROR_BAD_PEER]
The peer system is not recognized.

[GSK_ERROR_CLOSED]
The SSL session ended.

[GSK_AS400_ERROR_TIMED_OUT]
The value specified for the handshake timeout expired before the handshake completed.

[GSK_INSUFFICIENT_STORAGE]
Unable to allocate storage for the requested operation.

errnoValue
(Output) If operationCompleted is **QSOPOSTIOCOMPLETION**, **QSOSTARTSEND**, **QSOSTARTRECV** or **QSOSTARTACCEPT** and returnValue is negative, this field will contain an errno value further defining the error. This is also true if operationCompleted is **GSKSECURESOCSTARTINIT**, **GSKSECURESOCSTARTSEND** or **GSKSECURESOCSTARTRECV** and returnValue is GSK_ERROR_IO.

Possible values are:

[ECANCELED] The operation was cancelled by "QsoCancelOperation()—Cancel an I/O Operation" on page 80

If operationCompleted is QSOPOSTIOCOMPLETION:

[EAGAIN]	The specified timer value expired.
[ECLOSED]	The socket descriptor was closed before the timer expired.

If operationCompleted is QSOSTARTRECV or GSKSECURESOCSTARTRECV:

[EAGAIN]	The operation did not complete in the specified time.
[EIO]	Input/output error.
[ECONNABORTED]	Connection ended abnormally. This error code indicates that the transport provider ended the connection abnormally because of one of the following: • The retransmission limit has been reached for the data that was being sent on the socket. • A protocol error was detected.
[ECONNRESET]	A connection with a remote socket was reset by that socket.
[ECLOSED]	Connection was closed. Only valid for QSOSTARTRECV.
[EFAULT]	Read buffer pointer not valid.

If operationCompleted is QSOSTARTSEND or GSKSECURESOCSTARTSEND:

[EAGAIN]	The operation did not complete in the specified time.
[EIO]	Input/output error.
[EPIPE]	Broken pipe.
[ECLOSED]	Connection was closed. Only valid for QSOSTARTSEND
[EFAULT]	Send buffer pointer not valid.

If operationCompleted is QSOSTARTACCEPT:

[EAGAIN]	The operation did not complete in the specified time.
[ECONNABORTED]	Connection ended abnormally.
[ECLOSED]	Listening socket closed.
[EIO]	Input/output error.
[EMFILE]	Too many descriptors for this process.
[ENFILE]	Too many descriptors in system.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[EUNKNOWN]	Unknown system state.

If operationCompleted is GSKSECURESOCSTARTINIT:

[ECONNABORTED]	Connection ended abnormally.
[EDEADLK]	Resource deadlock avoided.
[EINTR]	Interrupted function call.
[EIO]	Input/output error.
[ETERM]	Operation terminated.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.

Any *errno* that can be returned by **send()** or **recv()** can be returned by this API if operationCompleted is **GSKSECURESOCSTARTINIT**. See "Sockets APIs," on page 1 for a description of the *errno* values they return.

If an *errno* is returned that is not in this list, see Errno Values for UNIX-Type Functions for a description of the *errno*.

operationCompleted (Output) The operation that was started and has now completed.

- 1 (QSOSTARTSEND)
- 2 (QSOSTARTRECV)
- 3 (QSOPOSTIOCOMPLETION)
- 4 (GSKSECURESOCSTARTSEND)
- 5 (GSKSECURESOCSTARTRECV)
- 6 (QSOSTARTACCEPT)
- 7 (GSKSECURESOCSTARTINIT)

secureDataTransferSize (Output) Number of bytes received or sent if operationCompleted is GSKSECURESOCSTARTRECV or GSKSECURESOCSTARTSEND respectively and returnValue equals GSK_OK.

bytesAvailable (Output) Number of bytes available to be read from connection. This parameter is valid only if operationCompleted is QSOSTARTACCEPT and returnValue is >= 0.

operationWaitTime (Output) The value of the operationWaitTime when the operation was started.

postedDescriptor (Output) Always set to negative one. This field is only used on input for QsoPostIOCompletion(). When the operation is retrieved with QsoWaitForIOCompletion(), the *descriptorHandle* should be used to identify the socket connection and not this field.

» *operationId* (Output) - An identifier to uniquely identify this operation or a group of operations. This value is preserved from the start operation and returned by QsoWaitForIOCompletion(). It is ignored unless "QsoCancelOperation()—Cancel an I/O Operation" on page 80 or "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86 is used. «

struct timeval * timeToWait

(Input) A pointer to a timeval structure that contains the time in seconds and microseconds for which the QsoWaitForIOCompletion() call should block if there is no completion status to receive.

If this parameter is null, QsoWaitForIOCompletion() waits indefinitely. If this value is specified, and 0 seconds 0 microseconds are specified, QsoWaitForIOCompletion() returns immediately.

Authorities

Authorization of *R (allow access to the object) to the certificate store file and its associated files is required. Authorization of *X (allow use of the object) to each directory of the path name of the certificate store file and its associated files is required.

Return Values

QsoWaitForIOCompletion returns an integer. Possible value are:

1 Completion of an overlapped I/O function has been returned.

-1 The QsoWaitForIOCompletion() function timed out or an error occurred. Errno value has been set.

0 If the QsoWaitForIOCompletion() function is issued with a timeToWait parameter that specifies 0 seconds 0 microseconds and there is no completion status to report, the function returns immediately with a return value of zero.

Errno Conditions

When QsoWaitForIOCompletion fails, errno can be set to one of the following:

[ETIME] The function has blocked for the time period specified and has no completion status to report.

[EFAULT]	Bad address. The system detected a bad address while attempting to access the completionStatus or the timeToWait parameter.
[EDESTROYED]	The I/O completion port has been destroyed.
[EINVAL]	The value of the I/O completion port is not valid or the timeToWait parameter is not valid.
[EINTR]	Interrupted function call.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPFA081 E	Unable to set return value or error code.
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.

Usage Notes

1. An errno of EDESTROYED indicates that the thread was waiting on the I/O completion port at the time that it was destroyed by another thread. When an I/O completion port is destroyed, all buffers that are associated with outstanding overlapped I/O operations are immediately available for use by the application program.
2. The application should first check the return value of the *QsoWaitForIOCompletion()* call to determine if the Qso_OverlappedIO_t structure specified by the completionStatus parameter has been updated. This structure is updated ONLY if the return value of the *QsoWaitForIOCompletion()* call is one (1).

Related Information

-  "QsoCancelOperation()—Cancel an I/O Operation" on page 80—Cancel an I/O Operation
- "QsoIsOperationPending()—Check if an I/O Operation is Pending" on page 86—Check if an I/O Operation is Pending

- "QsoCreateIOCompletionPort()—Create I/O Completion Port" on page 82—Create I/O Completion Port
- "QsoDestroyIOCompletionPort()—Destroy I/O Completion Port" on page 83—Destroy I/O Completion Port
- "QsoPostIOCompletion()—Post I/O Completion Request" on page 87—Post Request on I/O Completion Port
- "QsoStartAccept()—Start asynchronous accept operation" on page 90—Start asynchronous accept operation
- "QsoStartRecv()—Start Asynchronous Receive Operation" on page 94—Start Asynchronous Recv Operation
- "QsoStartSend()—Start Asynchronous Send Operation" on page 97—Start Asynchronous Send Operation
- gsk_secure_soc_startRecv()—Start Asynchronous Receive Operation on a secure session
- gsk_secure_soc_startSend()—Start Asynchronous Send Operation on a secure session
- gsk_secure_soc_startInit()—Start Asynchronous Operation to negotiate a secure session

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

Rbind()—Set Remote Address for Socket

BSD 4.3 Syntax

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

int Rbind(int socket_descriptor,
        struct sockaddr *local_address,
        int address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int Rbind(int socket_descriptor,
        const struct sockaddr *local_address,
        socklen_t address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

A program uses the *Rbind()* call to request that a SOCKS server allow an inbound connection request across a firewall. This call should only be used by applications that require inbound connections across a firewall, and should only be used for sockets with an address family of `af_inet`. Note that for an *Rbind()* call to succeed, a previous *connect()* call must have been issued for this thread, and must have resulted in an outbound connection over the same SOCKS server. The *Rbind()* inbound connection will be from the same IP address addressed by the original outbound connection. Caution must be exercised so that outbound and inbound connections over the SOCKS server are paired. In other words, all *Rbind()* inbound connections should immediately follow the outbound connection over the SOCKS server, and no intervening non-SOCKS connections relating to this thread can be attempted before the *Rbind()* runs. For an overview of using sockets and how to interact with a SOCKS server, see the topic about i5/OS client SOCKS support in the Sockets Programming in the iSeries Information Center.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket that is to be bound.

local_address
(Input) A pointer to a buffer of type **struct sockaddr** that contains the local address to which the socket is to be bound. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

address_length
(Input) The length of the *local_address*.

Authorities

- When the address type of the socket identified by the ***socket_descriptor*** is AF_INET, the thread must have retrieve, insert, delete, and update authority to the port specified by the ***local_address*** field. When the thread does not have this level of authority, an *errno* of EACCES is returned.
- When the address type of the socket identified by the ***socket_descriptor*** is AF_INET and is running IP over SNA, the thread must have retrieve, insert, delete, and update authority to the APPC device. When the thread does not have this level of authority, an *errno* of EACCES is returned.

Return Value

Rbind() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When an *Rbind()* fails, *errno* can be set to one of the following:

[EADDRNOTAVAIL] Address not available. This error code indicates one of the following:
- The SOCKS server specified is not reachable.
- The SOCKS server has denied the requested inbound connection.
- The Socket can no longer be used for an inbound connection.

[EAFNOSUPPORT] The type of socket is not supported in this protocol family.

The address family specified in the address structure pointed to by the *local_address* parameter cannot be used with the socket pointed to by the *socket_descriptor* parameter.

[EBADF] Descriptor not valid.

[EFAULT] Bad address.

The system detected an address that was not valid while attempting to access the *local_address* parameter.

[EINVAL] Parameter not valid. This error code indicates one of the following:
- The *address_length* parameter specifies a length that is negative or is not valid for the address family.
- The socket referenced by *socket_descriptor* is not a socket of type SOCK_RAW and is already bound to an address.
- The local address pointed to by the *local_address* parameter specified an address that was not valid.

[EIO] Input/output error.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTSOCK]	The specified descriptor does not reference a socket.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. If this call is issued for sockets with an address family other than `af_inet`, or if the thread has not performed an outbound connection through a SOCKS server, then a *bind()* call will be run instead. In this case the documented *errno* and usage notes for *bind()* apply.
2. The local IP address and port number specified for sockets with an address family of `af_inet` are ignored if *Rbind()* results in an inbound connection over a SOCKS server. In this scenario the socket is logically bound to the SOCKS server IP address coupled with a port selected via SOCKS server. If a *bind()* is performed, then the socket is bound to the local IP address and port number specified.
3. The *Rbind()* function may be explicitly used, or optionally you can compile your application with the __Rbind macro defined when you call the compiler. For example, if you are compiling with a Create C Module (CRTCMOD) CL command, specify __Rbind for the DEFINE keyword to cause the __Rbind macro to be defined before the compilation starts. Now all *bind()* calls in the program will become *Rbind()*. See **<sys/socket.h>** for a definition of the __Rbind macro.
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *Rbind()* API is mapped to *qso_Rbind98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "accept()—Wait for Connection Request and Make Connection" on page 4—Wait for Connection Request and Make Connection
- "bind()—Set Local Address for Socket" on page 13—Set Local Address for Socket
- "connect()—Establish Connection or Destination Address" on page 22—Establish Connection or Destination Address
- "getsockname()—Retrieve Local Address of Socket" on page 49—Retrieve Local Address of Socket

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

read()—Read from Descriptor

Syntax

```
#include <unistd.h>

ssize_t read(int file_descriptor,
             void *buf, size_t nbyte);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see “Usage Notes” on page 112.

From the file or socket indicated by *file_descriptor*, the **read()** function reads *nbyte* bytes of input into the memory area indicated by *buf*. If *nbyte* is zero, **read()** returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset) or any other type of file on which the job can do an **lseek()** operation, **read()** begins reading at the file offset associated with *file_descriptor*. A successful **read()** changes the file offset by the number of bytes read.

If **read()** is successful and *nbyte* is greater than zero, the access time for the file is updated.

read() is not supported for directories.

If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA specified, the data is read from the file assuming it is in textual form. The maximum number of bytes on a single read that can be supported for text data is 2,147,483,408 (2GB - 240) bytes. The data is converted from the code page of the file to the code page of the application, job, or system as follows:

- When reading from a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
- When reading from record files that are being used as stream files, end-of-line characters are added to the end of the data in each record.

There are some important considerations when the file is open for text conversion and the CCSIDs involved are not strictly single-byte:

- The **read()** will return the exact number of bytes requested. For some CCSIDs, this may mean that partial characters are returned at the end of the user buffer. In this case, the remainder of the character has been read from the file and internally buffered. The next consecutive read() will begin with the remainder of the partial character. However, if an **lseek()** is performed, the buffered data will be discarded. See lseek()—Set File Read/Write Offset for more information.
- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current file offset. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset may be 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved.

If O_TEXTDATA was not specified on the **open()**, the data is read from the file without conversion. The application is responsible for handling the data.

In the QSYS.LIB and independent ASP QSYS.LIB file systems, most end-of-file characters are symbolic; that is, they are stored outside the member. When reading:

- If O_TEXTDATA is specified, both symbolic and nonsymbolic end-of-file characters can be seen.
- If O_TEXTDATA is not specified (binary mode), only nonsymbolic end-of-file characters can be seen.

See the *Usage Notes* for “write()—Write to Descriptor” on page 185.

When *file_descriptor* refers to a socket, the **read()** function reads from the socket identified by the socket descriptor.

When attempting to read from an empty pipe or FIFO:

- If no job has the pipe or FIFO open for writing, **read()** return 0 to indicate end-of-file.

- If some job has the pipe or FIFO open for writing and O_NONBLOCK was specified, **read()** will fail and *errno* will be set to [EAGAIN].
- If some job has the pipe or FIFO open for writing and O_NONBLOCK was not specified, **read()** will block the calling thread until some data is written or until the pipe or FIFO is closed by all jobs that had the pipe or FIFO open for writing.

Parameters

file_descriptor
(Input) The descriptor to be read.

buf (Output) A pointer to a buffer in which the bytes read are placed.

nbyte (Input) The number of bytes to be read.

Authorities

No authorization is required.

Return Value

value **read()** was successful. The value returned is the number of bytes actually read and placed in *buf*. This number is less than or equal to *nbyte*. It is less than *nbyte* only if **read()** reached the end of the file before reading the requested number of bytes. If **read()** is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), **read()** places bytes containing zeros into *buf* in place of the unwritten bytes.

-1 **read()** was not successful. The *errno* global variable is set to indicate the error. If the value of *nbyte* is greater than SSIZE_MAX, **read()** sets *errno* to [EINVAL].

Error Conditions

If **read()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. This may occur if *file_descriptor* refers to a socket and the socket is using a connection-oriented transport service, and a *connect()* was previously completed. The thread, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the thread was not authorized to.
[EAGAIN]	If *file_descriptor* refers to a pipe or FIFO that has its O_NONBLOCK flag set, this error occurs if the **read()** would have blocked the calling thread.
[EBADF] *[EBADFID]* *[EBUSY]* *[EDAMAGE]* *[EFAULT]* *[EINTR]*	

Error condition	Additional information
[EINVAL]	This may occur if *file_descriptor* refers to a socket that is using a connectionless transport service, is not a socket of type SOCK_RAW, and is not bound to an address. The file resides in a file system that does not support large files, and the starting offset of the file exceeds 2GB minus 2 bytes.
[EIO]	
[ENOMEM]	
[ENOTAVAIL]	
[ENOTSAFE]	
[ENXIO]	
[EOVERFLOW]	The file is a regular file, *nbyte* is greater than 0, the starting offset is before the end-of-file, and the starting offset is greater than or equal to 2GB minus 2 bytes.
[ERESTART]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNABORTED]	This error code indicates that the transport provider ended the connection abnormally because of one of the following: • The retransmission limit has been reached for data that was being sent on the socket. • A protocol error was detected.
[ECONNREFUSED]	
[ECONNRESET]	
[EINTR]	
[ENOTCONN]	This error code is returned only on sockets that use a connection-oriented transport service.
[ETIMEDOUT]	A non-blocking **connect()** was previously completed that resulted in the connection timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.
[EUNATCH]	
[EWOULDBLOCK]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT]	
[EUNATCH]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – "Root" (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400
2. QSYS.LIB and Independent ASP QSYS.LIB File System Differences

   This function will fail with error code [ENOTSAFE] if the object on which this function is operation is a save file and multiple threads exist in the job.

   This function will fail with error code [EIO] if the file specified is a save file and the file does not contain complete save file data.

   The file access time for a database member is updated using the normal rules that apply to database files. At most, the access time is updated once per day.

   If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, you should avoid using other interfaces (such as the Source Entry Utility or database reads and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.
3. QOPT File System Differences

   The file access time is not updated on a **read()** operation.

   When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.
4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for

another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.)

Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. For sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM), a return value of zero indicates one of the following:
   - The partner program has issued a **close()** for the socket.
   - The partner program has issued a **shutdown()** to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A **shutdown()** to disable reading was previously done on the socket.
7. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM).
   - If a **connect()** has been issued previously, then data can be received only from the address specified in the previous **connect()**.
   - The address from which data is received is discarded, since the **read()** has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user supplied buffer, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     – The partner program has sent a NULL message (a datagram with no user data).
     – A **shutdown()** to disable reading was previously done on the socket.
     – The buffer length specified was zero.
8. For file systems that do not support large files, **read()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **read()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
9. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of zero. In addition, the access time for the file is updated.

Related Information

- The <**limits.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- creat()—Create or Rewrite File
- dup()—Duplicate Open File Descriptor
- dup2()—Duplicate Open File Descriptor to Another Descriptor
- fclear()—Write (Binary Zeros) to Descriptor
- fclear64()—Write (Binary Zeros) to Descriptor (Large File Enabled)
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 68—Perform I/O Control Request
- lseek()—Set File Read/Write Offset
- open()—Open File
- pread()—Read from Descriptor with Offset
- pread64()—Read from Descriptor with Offset (large file enabled)

- pwrite()—Write to Descriptor with Offset
- pwrite64()—Write to Descriptor with Offset (large file enabled)
- "readv()—Read from Descriptor Using Multiple Buffers"—Read from Descriptor Using Multiple Buffers
- "recv()—Receive Data" on page 119—Receive Data
- "recvfrom()—Receive Data" on page 122—Receive Data
- "recvmsg()—Receive a Message Over a Socket" on page 126—Receive Data or Descriptors or Both
- "write()—Write to Descriptor" on page 185—Write to Descriptor
- "writev()—Write to Descriptor Using Multiple Buffers" on page 192—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example opens a file and reads input:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main() {
  int ret, file_descriptor, rc;
  char buf[]="Test text";

  if ((file_descriptor = creat("test.output", S_IWUSR))!= 0)
     perror("creat() error");
  else {
    if (-1==(rc=write(file_descriptor, buf, sizof(buf)-1)))
       perror("write() error");
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }

  if ((file_descriptor = open("test.output", O_RDONLY)) < 0)
    perror("open() error");
  else {
    ret = read(file_descriptor, buf, sizeof(buf)-1));
    buf[ret] = 0x00;
    printf("block read: \n<%s>\", buf);
    if (close(file_descriptor)!= 0)
       perror("close() error");
  }
  if (unlink("test.output")!= 0)
     perror("unlink() error");
}
```

Output:

```
block read:
<Test text>
```

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

readv()—Read from Descriptor Using Multiple Buffers

Syntax

```
#include <sys/types.h>
#include <sys/uio.h>

int readv(int descriptor,
          struct iovec *io_vector[],
          int vector_length)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 117.

The *readv()* function is used to receive data from a file or socket descriptor. *readv()* provides a way for data to be stored in several different buffers (*scatter/gather I/O*).

See "read()—Read from Descriptor" on page 108 for more information related to reading from a descriptor.

Parameters

descriptor
(Input) The descriptor to be read. The descriptor refers to a file or a socket.

io_vector[]
(I/O) The pointer to an array of type **struct iovec**. **struct iovec** contains a sequence of pointers to buffers in which the data to be read is stored. The structure pointed to by the *io_vector* parameter is defined in **<sys/uio.h>**.

```
struct iovec {
   void      *iov_base;
   size_t   iov_len;
}
```

iov_base and *iov_len* are the only fields in *iovec* used by sockets. *iov_base* contains the pointer to a buffer and *iov_len* contains the buffer length. The rest of the fields are reserved.

vector_length
(Input) The number of entries in *io_vector*.

Authorities

No authorization is required.

Return Value

n **readv()** is successful, where n is the number of bytes read.

-1 **readv()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **readv()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. This may occur if *file_descriptor* refers to a socket and the socket is using a connection-oriented transport service, and a *connect()* was previously completed. The thread, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the thread was not authorized to.
[EAGAIN]	
[EBADF]	
[EBADFID]	
[EBUSY]	
[EDAMAGE]	
[EFAULT]	
[EINTR]	
[EINVAL]	This may occur if *file_descriptor* refers to a socket that is using a connectionless transport service, is not a socket of type `SOCK_RAW`, and is not bound to an address. The file resides in a file system that does not support large files, and the starting offset of the file exceeds 2 GB minus 2 bytes.
[EIO]	
[ENOMEM]	
[ENOTAVAIL]	
[ENOTSAFE]	
[EOVERFLOW]	The file is a regular file, *nbyte* is greater than 0, the starting offset is before the end-of-file and is greater than or equal to 2GB minus 2 bytes.
[ERESTART]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNABORTED]	This error code indicates that the transport provider ended the connection abnormally because of one of the following: • The retransmission limit has been reached for data that was being sent on the socket. • A protocol error was detected.
[ECONNREFUSED]	
[ECONNRESET]	
[EINTR]	
[ENOTCONN]	
[ETIMEDOUT]	A non-blocking **connect()** was previously completed that resulted in the connection timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.
[EUNATCH]	
[EWOULDBLOCK]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT]	
[EUNATCH]	

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – ″Root″ (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400
2. The *io_vector*[] parameter is an array of **struct iovec** structures. When a *readv()* is issued, the system processes the array elements one at a time, starting with *io_vector*[0]. For each element, **iov_len** bytes of received data are placed in storage pointed to by **iov_base**. Data is placed in storage until all buffers are full, or until there is no more data to receive. Only the storage pointed to by **iov_base** is updated. No change is made to the **iov_len** fields. To determine the end of the data, the application program must use the following:

   - The function return value (the total number of bytes received).

- The lengths of the buffers pointed to by **iov_base**.

3. For sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM), a returned value of zero indicates one of the following:

   - The partner program has issued a *close()* for the socket.
   - The partner program has issued a *shutdown()* to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A *shutdown()* to disable reading was previously done on the socket.

4. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM):
   - If a *connect()* has been issued previously, then data can be received only from the address specified in the previous *connect()*.
   - The address from which data is received is discarded, because the *readv()* has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user-supplied buffers, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     – The partner program has sent a NULL message (a datagram with no user data).
     – A *shutdown()* to disable reading was previously done on the socket.
     – The buffer length specified by the application was zero.

5. For the file systems that do not support large files, **readv()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **readv()** will return [EOVERFLOW] if the starting offset exceeds 2GB minus 2 bytes and file was not opened for large file access.
6. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.
7. QOPT File System Differences

   When reading from files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being read are ignored.
8. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of 0. In addition, the access time for the file is updated.

Related Information

- The <**limits.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- creat()—Create or Rewrite File
- dup()—Duplicate Open File Descriptor
- dup2()—Duplicate Open File Descriptor to Another Descriptor
- fclear()—Write (Binary Zeros) to Descriptor
- fclear64()—Write (Binary Zeros) to Descriptor (Large File Enabled)
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 68—Perform I/O Control Request
- lseek()—Set File Read/Write Offset
- open()—Open File
- “read()—Read from Descriptor” on page 108—Read from Descriptor
- “recv()—Receive Data” on page 119—Receive Data
- “recvfrom()—Receive Data” on page 122—Receive Data

- "recvmsg()—Receive a Message Over a Socket" on page 126—Receive Data or Descriptors or Both
- "write()—Write to Descriptor" on page 185—Write to Descriptor
- "writev()—Write to Descriptor Using Multiple Buffers" on page 192—Write to Descriptor Using Multiple Buffers

API introduced: V3R1

---

recv()—Receive Data

BSD 4.3 Syntax

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

int recv(int socket_descriptor,
        char *buffer,
        int buffer_length,
        int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t recv(int socket_descriptor,
        void *buffer,
        size_t buffer_length,
        int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *recv()* function is used to receive data through a socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The socket descriptor that is to be read from.

buffer (Input) The pointer to the buffer in which the data that is to be read is stored.

buffer_length
(Input) The length of the *buffer*.

flags (Input) A flag value that controls the reception of the data. The *flags* value is either zero, or is obtained by performing an OR operation on one or more of the following constants:

MSG_OOB Receive out-of-band data. Valid only for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

MSG_PEEK	Obtain a copy of the message without removing the message from the socket.
MSG_WAITALL	Wait for a full request or an error.

Authorities

No authorization is required.

Return Value

recv() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes received.

Error Conditions

When *recv()* fails, *errno* can be set to one of the following:

[EACCES]	Permission denied.
	The socket pointed to by the *socket_descriptor* parameter is using a connection-oriented transport service, and a *connect()* was previously completed. The process, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the process was not authorized to.
[EBADF]	Descriptor not valid.
[ECONNABORTED]	Connection ended abnormally.
	This error code indicates that the transport provider ended the connection abnormally because of one of the following: • The retransmission limit has been reached for data that was being sent on the socket. • A protocol error was detected.
[ECONNREFUSED]	The destination socket refused an attempted connect operation.
[ECONNRESET]	A connection with a remote socket was reset by that socket.
[EFAULT]	Bad address.
	The system detected an address which was not valid while attempting to access the *buffer* parameter.
[EINTR]	Interrupted function call.
[EINVAL]	Parameter not valid.
	This error code indicates one of the following: • The *buffer_length* parameter specifies a negative value. • The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but no OOB data was available to be received. • The *flags* parameter specifies a value that includes the `MSG_OOB` flag, and the socket option `SO_OOBINLINE` has been set. • The *socket_descriptor* parameter points to a socket that is using a connectionless transport service, is not a socket of type `SOCK_RAW`, and is not bound to an address.
[EIO]	Input/output error.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[ENOTCONN]	Requested operation requires a connection.
	This error code is returned only on sockets that use a connection-oriented transport service.
[ENOTSOCK]	The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *flags* parameter specifies a value that includes the MSG_OOB flag, but the *socket_descriptor* parameter points to a connectionless socket.
- The *flags* parameter specifies a value that includes the MSG_OOB flag, but the *socket_descriptor* parameter points to a socket that does not have an address family of AF_INET or AF_INET6.

[ETIMEDOUT] A remote host did not respond within the timeout period.

A nonblocking *connect()* call was previously done that resulted in the connection establishment timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. For sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM), a returned value of zero indicates one of the following:
   - The partner program has issued a *close()* for the socket.
   - The partner program has issued a *shutdown()* to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A *shutdown()* to disable reading was previously done on the socket.
2. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM):
   - If a *connect()* has been issued previously, then data can be received only from the address specified in the previous *connect()*.
   - The address from which data is received is discarded, since the *recv()* has no address parameter.
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user supplied buffer, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     – The partner program has sent a NULL message (a datagram with no user data),
     – A *shutdown()* to disable reading was previously done on the socket.
     – The buffer length specified was zero.
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *recv()* API is mapped to *qso_recv98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- "recvfrom()—Receive Data"—Receive Data
- "recvmsg()—Receive a Message Over a Socket" on page 126—Receive Data or Descriptors or Both

API introduced: V3R1

---

recvfrom()—Receive Data

BSD 4.3 Syntax

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

int recvfrom(int socket_descriptor,
             char *buffer,
             int buffer_length,
             int flags,
             struct sockaddr *from_address,
             int *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t recvfrom(int socket_descriptor,
             void *buffer,
             size_t buffer_length,
             int flags,
             struct sockaddr *from_address,
             socklen_t *address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *recvfrom()* function is used to receive data through a connected or unconnected socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The socket descriptor that is to be read from.

buffer (Input) The pointer to the buffer in which the data that is to be read is stored.

buffer_length
(Input) The length of the *buffer*.

int flags
(Input) A flag value that controls the reception of the data. The *flags* value is either zero, or is obtained by performing an OR operation on one or more of the following constants:

MSG_OOB	Receive out-of-band data. Valid only for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
MSG_PEEK	Obtain a copy of the message without removing the message from the socket.
MSG_WAITALL	Wait for a full request or an error.

from_address
(Output) A pointer to a buffer of type **struct sockaddr** that contains the address from which the message was received.

The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

Note: See the usage notes about using different address families with **sockaddr_storage**.

address_length
(Input/output) This parameter is a value-result field. The caller passes a pointer to the length of the *from_address* parameter. On return from the call, *address_length* will contain the actual length of the address.

Authorities

An *errno* of EACCES is returned when the socket pointed to by the ***socket_descriptor*** field is address family AF_INET and a nonblocking connect was attempted previously and was not successful. The nonblocking connect was not successful because the thread did not have authority to the associated APPC device. The thread performing the nonblocking connect must have retrieve, insert, delete, and update authority to the APPC device.

Return Value

recvfrom() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes received.

Error Conditions

When *recvfrom()* fails, *errno* can be set to one of the following:

[EACCES] Permission denied.

The socket pointed to by the *socket_descriptor* parameter is using a connection-oriented transport service, and a *connect()* was previously completed. The process, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the process was not authorized to.

[EBADF] Descriptor not valid.

[ECONNABORTED] Connection ended abnormally.

This error code indicates that the transport provider ended the connection abnormally because of one of the following:

- The retransmission limit has been reached for data that was being sent on the socket.
- A protocol error was detected.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

[ECONNRESET] A connection with a remote socket was reset by that socket.

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *buffer*, *from_address*, or *address_length* parameter.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *buffer_length* parameter specifies a negative value.
- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but no OOB data was available to be received.
- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, and the socket option `SO_OOBINLINE` has been set.
- The *socket_descriptor* parameter points to a socket that is using a connectionless transport service, is not a socket of type `SOCK_RAW`, and is not bound to an address.

[EIO] Input/output error.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTCONN] Requested operation requires a connection.

This error code is returned only on sockets that use a connection-oriented transport service.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but the *socket_descriptor* parameter points to a connectionless socket.
- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but the *socket_descriptor* parameter points to a socket that does not have an address family of `AF_INET` or `AF_INET6`.

[ETIMEDOUT] A remote host did not respond within the timeout period.

A non-blocking *connect()* was previously issued that resulted in the connection establishment timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. For sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM), a returned value of zero indicates one of the following:
   - The partner program has issued a *close()* for the socket.
   - The partner program has issued a *shutdown()* to disable writing to the socket.
   - The connection is broken and the error was returned on a previously issued socket function.
   - A *shutdown()* to disable reading was previously done on the socket.
2. If the socket is using a connection-oriented transport service, the *from_address* and *address_length* parameters are ignored.
3. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM):
   - If a *connect()* has been issued previously, then data can be received only from the address specified in the previous *connect()*.
   - If the *from_address* parameter is set to NULL or *address_length* specifies a value of zero, the address from which data is received is discarded by the system.
   - If the length of the address to be returned exceeds the length of the *from_address* parameter, the returned address is truncated.
   - The structure **sockaddr** is a generic structure used for any address family but it is only 16 bytes long. The actual address returned for some address families may be much larger. You should declare storage for the address with the structure **sockaddr_storage**. This structure is large enough and aligned for any protocol-specific structure. It may then be cast as **sockaddr** structure for use on the APIs. The *ss_family* field of the **sockaddr_storage** will always align with the family field of any protocol-specific structure.

     The BSD 4.3 structure is:

```
#define _SS_MAXSIZE 304
#define _SS_ALIGNSIZE (sizeof (char*))
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - sizeof(sa_family_t))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))

struct sockaddr_storage {
    sa_family_t   ss_family;
    char         _ss_pad1[_SS_PAD1SIZE];
    char*        _ss_align;
    char         _ss_pad2[_SS_PAD2SIZE];
};
```

     The BSD 4.4/UNIX 98 compatible structure is:

```
#define _SS_MAXSIZE 304
#define _SS_ALIGNSIZE (sizeof (char*))
#define _SS_PAD1SIZE (_SS_ALIGNSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)))
#define _SS_PAD2SIZE (_SS_MAXSIZE - (sizeof(uint8_t) + sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))
```

```
struct sockaddr_storage {
    uint8_t       ss_len;
    sa_family_t   ss_family;
    char         _ss_pad1[_SS_PAD1SIZE];
    char*        _ss_align;
    char         _ss_pad2[_SS_PAD2SIZE];
};
```

- If the socket is using an address family of AF_UNIX, the address (which is a path name) is returned in the default coded character set identifier (CCSID) currently in effect for the job.
- If the socket is using an address family of AF_UNIX_CCSID, the output structure sockaddr_unc defines the format and coded character set identifier (CCSID) of the address (which is a path name).
- The entire message must be read in a single read operation. If the size of the message is too large to fit in the user supplied buffer, the remaining bytes of the message are discarded.
- A returned value of zero indicates one of the following:
  - The partner program has sent a NULL message (a datagram with no user data).
  - A *shutdown()* to disable reading was previously done on the socket.
  - The buffer length specified was zero.

4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *recvfrom()* API is mapped to *qso_recvfrom98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- "recv()—Receive Data" on page 119—Receive Data
- "recvmsg()—Receive a Message Over a Socket"—Receive Data or Descriptors or Both

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

recvmsg()—Receive a Message Over a Socket

BSD 4.3 Syntax

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

int recvmsg(int socket_descriptor,
            struct msghdr *message_structure,
            int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t recvmsg(int socket_descriptor,
                struct msghdr *message_structure,
                int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *recvmsg()* function is used to receive data or descriptors or both through a connected or unconnected socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The socket descriptor that is to be read from.

message_structure
(I/O) The pointer to the message structure that contains the following:

- The address from which the message was received
- The vector array in which the data received is stored
- The ancillary data/access rights list in which the received descriptors are stored

The structure pointed to by the *message_structure* parameter is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct msghdr {
  caddr_t       msg_name;
  int           msg_namelen;
  struct iovec *msg_iov;
  int           msg_iovlen;
  caddr_t       msg_accrights;
  int           msg_accrightslen;
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
struct msghdr {
  void         *msg_name;
  socklen_t     msg_namelen;
  struct iovec *msg_iov;
  int           msg_iovlen;
  void         *msg_control;      /* Set to NULL if not needed  */
  socklen_t     msg_controllen;   /* Set to 0 if not needed     */
  int           msg_flags;
};
```

The *msg_name* and *msg_namelen* fields contain the address and address length to which the message is sent. For further information on the structure of socket addresses, see Sockets Programming in the iSeries Information Center. If the *msg_name* field is set to a NULL pointer, the address information is not returned.

The *msg_iov* and *msg_iovlen* fields are for scatter/gather I/O.

The BSD 4.3 structure uses the *msg_accrights* and *msg_accrightslen* fields to pass descriptors. The *msg_accrights* field is a list of zero or more descriptors, and *msg_accrightslen* is the total length (in bytes) of the descriptor list.

The BSD 4.4/UNIX 98 compatible structure uses the *msg_control* and *msg_controllen* fields to pass ancillary data. The *msg_control* field is a pointer to ancillary data (of length *msg_controllen*) with the form:

```
struct cmsghdr {
      socklen_t cmsg_len;    /* # bytes, including this header */
      int       cmsg_level;  /* originating protocol           */
      int       cmsg_type;   /* protocol-specific type         */
                    /* followed by unsigned char cmsg_data[];  */
};
```

The *cmsg_len* field is the total length including this header. *cmsg_level* is the originating protocol. *cmsg_type* is the protocol-specific type. If ancillary data is not being passed, the *msg_control* field must be initalized to NULL and the *msg_controllen* field must be initialized to 0. The following table lists the supported ancillary data types when using the BSD 4.4/UNIX 98 compatible structures.

Ancillary Data Types That Apply to the Socket Layer (where *cmsg_level* is SOL_SOCKET):

cmsg_type	cmsg_data
SCM_RIGHTS	The rest of the buffer is a list of zero or more descriptors received. This ancillary data type is only supported for sockets with an address family of AF_UNIX or AF_UNIX_CCSID.

Macros are provided for navigating these structures.

- *CMSG_DATA(cmsg)* If the argument is a pointer to a *cmsghdr* structure, this macro returns an unsigned character pointer to the data array associated with the *cmsghdr* structure.
- *CMSG_NXTHDR(mhdr,cmsg)* If the first argument is a pointer to a *msghdr* structure and the second argument is a pointer to a *cmsghdr* structure in the ancillary data, pointed to by the msg_control field of that msghdr structure, this macro returns a pointer to the next *cmsghdr* structure, or a null pointer if this structure is the last *cmsghdr* in the ancillary data.
- *CMSG_FIRSTHDR(mhdr)* If the argument is a pointer to a *msghdr* structure, this macro returns a pointer to the first *cmsghdr* structure in the ancillary data associated with this *msghdr* structure, or a null pointer if there is no ancillary data associated with the *msghdr* structure.

The BSD 4.4/UNIX 98 compatible structure has the *msg_flags* for message level flags including:

- *MSG_TRUNC* Message data was truncated
- *MSG_CTRUNC* Ancillary data was truncated.
- *MSG_EOR* End of record (if supported by the protocol).
- *MSG_OOB* Out-of-band data.

flags (Input) A flag value that controls the reception of the data. The *flags* value is either zero, or is obtained by performing an OR operation on one or more of the following constants:

MSG_OOB	Receive out-of-band data. Valid only for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
MSG_PEEK	Obtain a copy of the message without removing the message from the socket.
MSG_WAITALL	Wait for a full request or an error.

Authorities

- An *errno* of EACCES is returned when the socket pointed to by the ***socket_descriptor*** field is address family AF_INET and a nonblocking connect was attempted previously and was not successful. The nonblocking connect was not successful because the thread did not have authority to the associated APPC device. The thread performing the nonblocking connect must have retrieve, insert, delete, and update authority to the APPC device.

- If this thread is receiving socket descriptors, it must have *ALLOBJ special authority or must be running under the same user profile as the thread that sent the descriptors using sendmsg. If both of these conditions are not true, the descriptors are reclaimed by the machine and an *errno* of EACCES is returned.

Return Value

recvmsg() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes received.

Error Conditions

When *recvmsg()* fails, *errno* can be set to one of the following:

[EACCES] Permission denied.

The socket pointed to by the *socket_descriptor* parameter is using a connection-oriented transport service, and a *connect()* was previously completed. The process, however, does not have the appropriate privileges to the objects that were needed to establish a connection. For example, the *connect()* required the use of an APPC device that the process was not authorized to.

If the *msg_accrights* and *msg_accrightslen* fields (or the BSD 4.4/UNIX 98 compatible fields *msg_control* and *msg_controllen*) were specified and descriptors were sent, this error indicates that this job does not have the appropriate privileges required to receive the descriptor. When this occurs, the descriptor is reclaimed by the system and the resource that it represented is closed.

[EBADF] Descriptor not valid.

[ECONNABORTED] Connection ended abnormally.

This error code indicates that the transport provider ended the connection abnormally because of one of the following:

- The retransmission limit has been reached for data that was being sent on the socket.
- A protocol error was detected.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

[ECONNRESET] A connection with a remote socket was reset by that socket.

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *message_structure* parameter or a field within the structure pointed to by the *message_structure* parameter.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *msg_iovlen* field or the **iov_len** field in a **iovec** structure specifies a negative value.
- The *flags* parameter specifies a value that includes the MSG_OOB flag, but no OOB data was available to be received.
- The *flags* parameter specifies a value that includes the MSG_OOB flag, and the socket option SO_OOBINLINE has been set.
- The *socket_descriptor* parameter points to a socket that is using a connectionless transport service, is not a socket of type SOCK_RAW, and is not bound to an address.
- The *msg_accrightslen* field in the *msghdr* structure specifies a negative value or is not large enough when *msg_accrights* was specified.
- The *msg_controllen* field in the *msghdr* structure specifies a negative value or is not large enough when *msg_control* was specified.

[EIO] Input/output error.

[EMFILE] Too many descriptions for this process.

[EMSGSIZE] Message size out of range.

The *msg_iovlen* field specifies a value that is greater than [MSG_MAXIOVLEN] (defined in **<sys/socket.h>**).

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTCONN] Requested operation requires a connection.

This error code is returned only on sockets that use a connection-oriented transport service.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *flags* parameter specifies a value that includes the MSG_OOB flag, but the *socket_descriptor* parameter points to a connectionless socket.
- The *flags* parameter specifies a value that includes the MSG_OOB flag, but the *socket_descriptor* parameter points to a socket that does not have an address family of AF_INET or AF_INET6.

[ETIMEDOUT] A remote host did not respond within the timeout period.

A non-blocking *connect()* was previously issued that resulted in the connection establishment timing out. No connection is established. This error code is returned only on sockets that use a connection-oriented transport service.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The following applies to sockets that use a connection-oriented transport service (for example, sockets with a type of SOCK_STREAM),
   - The *msg_name* and *msg_namelen* fields in the structure pointed to by the *message_structure* parameter are ignored.
   - A returned value of zero indicates one of the following:
     – The partner program has issued a *close()* for the socket.
     – The partner program has issued a *shutdown()* to disable writing to the socket.
     – The connection is broken and the error was returned on a previously issued socket function.
     – A *shutdown()* to disable reading was previously done on the socket.
2. The following applies to sockets that use a connectionless transport service (for example, a socket with a type of SOCK_DGRAM):
   - If a *connect()* has been issued previously, then data can be received only from the address specified in the previous *connect()*.
   - If the *msg_name* field is set to NULL or *msg_namelen* field specifies a value of zero, the address from which data is received is discarded.
   - If the length of the address to be returned exceeds the length specified by the *msg_namelen* field, the returned address is truncated.
   - If the socket is using an address family of AF_UNIX, the address (which is a path name) is returned in the default coded character set identifier (CCSID) currently in effect for the job.
   - If the socket is using an address family of AF_UNIX_CCSID, the output structure sockaddr_unc defines the format and coded character set identifier (CCSID) of the address (which is a path name).
   - The entire message must be read in a single read operation. If the size of the message is too large to fit in the user supplied buffer, the remaining bytes of the message are discarded.
   - A returned value of zero indicates one of the following:
     – The partner program has sent a NULL message (a datagram with no user data).
     – A *shutdown()* to disable reading was previously done on the socket.
     – The buffer length specified was zero.
3. The passing of descriptors is only supported over sockets that have an address family of AF_UNIX or AF_UNIX_CCSID. The *msg_accrightslen* and the *msg_accrights* fields (or the BSD 4.4/UNIX 98 compatible fields *msg_control* and *msg_controllen*) are ignored if the socket has any other address family. The value of *msg_accrightslen* (or the BSD 4.4/UNIX 98 compatible field *msg_controllen*) should be checked to determine if a descriptor has been returned. When you use *sendmsg()* and *recvmsg()* to pass descriptors, the target job must be running with either of the following:

   - The same user profile as the source job (in essence, passing the descriptor to yourself)
   - *ALLOBJ special authority

   If the target job closes the receiving end of the UNIX domain socket while a descriptor is in transit, the descriptor is reclaimed by the system, and the resource that it represented is closed. For files and directories, the ability to pass descriptors using *sendmsg()* and *recvmsg()* is only supported for objects in the Root, QOpenSys, User-defined file systems (UDFS), and Network File System (NFS).

   Note: The *recvmsg()* API will not block unless a data buffer is specified.
4. *recvmsg()* accepts a pointer to an array of **iovec** structures in the **msghdr** structure. The *msg_iovlen* field is used to determine the number of elements in the array (the number of **iovec** structures specified). When *recvmsg()* is issued, the system processes the array elements one at a time, starting with the first structure. For each element of the array (for each structure), *iov_len* bytes of received data are placed in storage pointed to by *iov_base*. Data is placed in storage until all buffers are full, or

until there is no more data to receive. Only the memory pointed to by *iov_base* is updated. No change is made to the *iov_len* fields. To determine the end of the data, the application program must use the following:

- The function return value (the total number of bytes received).
- The lengths of the buffers pointed to by *iov_base*.

5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *recvmsg()* API is mapped to *qso_recvmsg98()*.
6. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ENOTSUP]. See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.
7. When the descriptor is obtained using *recvmsg()*, any information accessed using that descriptor with the various read and write interfaces will be in binary, even if the original descriptor's accesses would have had text conversions occur. See Using CCSIDs and code pages in the open—Open file documentation for more information on text conversion.

Related Information

- For additional information and sample programs on how to use sendmsg() and recvmsg() to pass descriptors between system jobs, see Sockets Programming in the iSeries Information Center.
- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- "givedescriptor()—Pass Descriptor Access to Another Job" on page 60—Pass Descriptor Access to Another Job
- "recv()—Receive Data" on page 119—Receive Data
- "recvfrom()—Receive Data" on page 122—Receive Data
- "takedescriptor()—Receive Socket Access from Another Job" on page 183—Receive Descriptor Access from Another Job

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

rexec()—Issue a Command on a Remote Host

Syntax

```
#include <arpa/rexec.h>

int rexec(char **host,
          int port,
          char *user,
          char *password,
          char *command,
          int *errorDescriptor);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The **rexec()** function is used to open a connection to a remote host and send a user ID, password, and command to the remote host. The remote host verifies that the user ID and password are valid. The command is issued after the user ID and password are validated.

Parameters

host (Input)
A pointer to a character string that identifies the name of a remote *host*.

port (Input)
The well-known Internet *port* to use for the connection. A pointer to the structure containing the necessary *port* can be obtained by issuing the following call:

```
getservbyname("exec", "tcp");
```

The *port* returned by *getservbyname()* is the port on which the remote *host* is listening for incoming *rexec()* connections.

user (Input)
A character string that identifies a valid *user* on the remote *host*.

password (Input)
A character string that identifies the *password* for the *user* on the remote *host*. Specify a value of NULL if password security is not active on the remote host.

command (Input)
A character string that identifies the *command* to be issued on the remote *host*.

errorDescriptor (Input/Output)
One of the following values:

non-NULL A second connection is set up and that a descriptor for it is placed in the *errorDescriptor* parameter. This connection provides standard error results of the remote *command*. This information also includes remote authorization failure if *rexec()* is unsuccessful.

NULL The standard error results of the remote *command* are the same as the standard output return value.

Return Value

rexec() returns an integer. Possible values are:

Non-negative
(successful) A socket to the remote command is returned and can be used to receive results of running the command on the remote host.

- If errorDescriptor is non-NULL, standard error results of running the command on the remote host can be received by using the errorDescriptor.
- If errorDescriptor is NULL, standard error results of running the command on the remote host can be received with the standard output results by using the return value from rexec().

[-1] (unsuccessful) Refer to errno for a description of the failure.

- If errno is 0 and errorDescriptor is NULL, the host does not exist or remote authorization failed.
- If errno is 0 and errorDescriptor is -1, the host does not exist.
- If errno is 0 and errorDescriptor is non-negative, remote authorization failed.

Authorities

No authorization is required.

Error Conditions

When the **rexec()** API fails, errno can be set to one of following:

[ECONNABORTED] Connection ended abnormally.

[ECONNREFUSED]	The destination socket refused an attempted connect operation. This error occurs when the rexec server on the remote system is not active.
[ECONNRESET]	A connection with a remote socket was reset by that socket.
[EFAULT]	Bad address. System detected an address which was not valid while attempting to access the address parameters.
[EHOSTUNREACH]	A route to the remote host is not available.
[EINTR]	Interrupted function call.
[EINVAL]	Parameter not valid.
[EMFILE]	Too many descriptors for this process.
[ENFILE]	Too many descriptors in system.
[EPIPE]	Broken pipe.
[ETIMEDOUT]	A remote host did not respond within the timeout period. This error code is returned when connection establishment times out. No connection is established. A possible cause may be that the partner application is bound, but the partner application has not yet issued a listen().
[EUNATCH]	The protocol required to support address family AF_INET, is not available at this time.
[EUNKNOWN]	Unknown system state.

Usage Notes

- The password does not get encrypted while sent to the rexec server.
- Any results of the command received by the caller of rexec() are not converted from CCSID 819. Conversion from ASCII ccsid 819 to the CCSID of the process or thread is the caller's responsibility.
- If a remote authorization failure occurs, the return value will be -1 and if errorDescriptor is non-null a message indicating the authorization failure can be received with the socket descriptor from errorDescriptor.
- Any socket descriptor returned to the caller of rexec() must be explicitly closed by the caller.
- The user, password, and command will be translated from the job ccsid to ASCII ccsid 819 to be sent to the remote host.
- Issuing rexec() to a remote host that is configured to set up a SOCKSified connection is not supported.

Related Information

- "rexec_r()—Issue a Command on a Remote Host" on page 136—Issue a Command on a Remote Host

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **rexec()** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <qtqiconv.h>
#include <arpa/rexec.h>
#include <errno.h>

#define BufLen 256

void main()
{
  int sd = -1, rc;
```

```
  int responseLen = BufLen;
  int outbytesleft = BufLen;
  int bytesRead, saveBytesRead;
  struct servent serv_ent;
  struct servent_data serv_ent_data;
  char inbuf[BufLen];
  char outbuf[BufLen];
  char *inbufPtr = (char *)inbuf;
  char *outbufPtr = (char *)outbuf;
  iconv_t cd;
  QtqCode_T toCode   = {0,0,0,0,0,0};    /* Convert to job CCSID */
  QtqCode_T fromCode = {819,0,0,1,0,0};  /* ASCII CCSID */
  char *host;
  char remoteHost[256] = "remoteHost";
  char user[32] = "userName";
  char password[32] = "myPassword";
  char cmd[256] = "commandToRun";
  int *errordesc = NULL;

  /* Must zero this out before call or results will be unpredictable. */
  memset(&serv_ent_data.serve_control_blk, 0x00, sizeof(struct netdb_control_block));

  /* retrieve the rexec server port number */
  rc = getservbyname_r("exec", "tcp", &serv_ent, &serv_ent_data);
  if (rc < 0)
    printf("getservbyname_r() failed with errno = %d\n",errno);

  host = remoteHost;
  errno = 0;

  /* Issue the rexec API */
  sd = rexec(&host, serv_ent.s_port, user, password, cmd, errordesc);
  if (sd == -1) /* check if rexec() failed */
  {
    if (errno)
      printf("rexec() failed with errno = %d\n",errno);
    else
      printf("Either the host does not exist or remote authentication failed.\n");
  }
  else /* rexec() was successful */
  {
    bytesRead = recv(sd, inbuf, responseLen, 0);
    if (bytesRead > 0)
    {
      saveBytesRead = bytesRead;
      inbuf[bytesRead-1] = 0;  /* Null terminate */
      /* translate from ASCII to EBCDIC */
      cd = QtqIconvOpen(&toCode, &fromCode);
      iconv(cd,
            (unsigned char **)&inbufPtr,
            (unsigned int *)&bytesRead,
            (unsigned char **)&outbufPtr,
            (unsigned int *)&outbytesleft);
      iconv_close(cd);
      outbufPtr -= saveBytesRead;  /* Reset the buffer pointers */
      printf("%s\n",outbufPtr);
    }
    else if (bytesRead == 0)
      printf("The remote host closed the connection.\n");
    else
      printf("recv() failed with errno = %d\n",errno);
  }
  if (sd != -1)
    close(sd);  /* close the connection. */
  return;
}
```

API introduced: V5R1

rexec_r()—Issue a Command on a Remote Host

Syntax

```
#include <arpa/rexec.h>

int rexec_r(char **host,
            int port,
            char *user,
            char *password,
            char *command,
            int *errorDescriptor,
            struct hostent_data *hostEntData);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The **rexec_r()** function is used to open a connection to a remote host and send a user ID, password, and command to the remote host. The remote host verifies that the user ID and password are valid. The command will be issued after the user ID and password are validated.

Parameters

host (Input)
A pointer to a character string that identifies the name of a remote *host*.

port (Input)
The well-known Internet *port* to use for the connection. A pointer to the structure that contains the necessary *port* can be obtained by issuing the following call:

```
struct servent servEnt;
struct servent_data servEntData;
memset(&servEntData.serve_control_blk, 0x00, sizeof(struct netdb_control_block));
getservbyname_r("exec", "tcp", &servEnt, &servEntData);
```

The *port* returned by *getservbyname_r()* is the port that the remote *host* is listening on for incoming *rexec_r()* connections.

user (Input)
A character string that identifies a valid *user* on the remote *host*.

password (Input)
A character string that identifies the *password* for the *user* on the remote *host*. Specify a value of NULL if password security is not active on the remote host.

command (Input)
A character string that identifies the *command* to be issued on the remote *host*.

errorDescriptor (Input/Output)
One of the following values:

non-NULL A second connection is set up, and a descriptor for it is placed in the *errorDescriptor* parameter. This connection provides standard error results of the remote *command*. This information will also include remote authorization failure if *rexec()* is unsuccessful.

NULL The standard error results of the remote *command* is the same as the standard output return value.

hostEntData (Input/Output)

A pointer to the hostent_data structure, which is used to pass and preserve results between function calls. *rexec_r()* performs a *gethostbyname_r()* and each thread needs its own host data. The field *host_control_block* in the *hostent_data* structure must be initialized to hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire *hostent_data* structure must be initialized to hexadecimal zeros before its initial use. The *hostent_data* structure is defined in <netdb.h>>.

Return Value

rexec_r() returns an integer. Possible values are:

Non-negative

(successful) A socket to the remote command is returned and can be used to receive results of running the command on the remote host.

- If errorDescriptor is non-NULL, standard error results of running the command on the remote host can be received by using the errorDescriptor.
- If errorDescriptor is NULL, standard error results of running the command on the remote host can be received along with the standard output results by using the return value from rexec_r().

[-1] (unsuccessful) Refer to errno for a description of the failure.

- If errno is 0 and errorDescriptor is NULL, the host does not exist or remote authorization failed.
- If errno is 0 and errorDescriptor is -1, the host does not exist.
- If errno is 0 and errorDescriptor is Non-negative, remote authorization failed.

Authorities

No authorization is required.

Error Conditions

When the **rexec_r()** API fails, errno can be set to one of following:

[ECONNABORTED]	Connection ended abnormally.
[ECONNREFUSED]	The destination socket refused an attempted connect operation. This error occurs when the rexec server on the remote system is not active.
[ECONNRESET]	A connection with a remote socket was reset by that socket.
[EFAULT]	Bad address. System detected an address which was not valid while attempting to access the address parameters.
[EHOSTUNREACH]	A route to the remote host is not available.
[EINTR]	Interrupted function call.
[EINVAL]	Parameter not valid. This error code occurs when the hostEntData structure has not been initialized to hexadecimal zeros. For corrective action, see the description for structure hostent_data.
[EMFILE]	Too many descriptors for this process.
[ENFILE]	Too many descriptors in system.
[EPIPE]	Broken pipe.
[ETIMEDOUT]	A remote host did not respond within the timeout period. This error code is returned when connection establishment times out. No connection is established. A possible cause may be that the partner application is bound, but the partner application has not yet issued a listen().
[EUNATCH]	The protocol required to support address family AF_INET, is not available at this time.

[EUNKNOWN] Unknown system state.

Usage Notes

- The password does not get encrypted while sent to the rexec server.
- Any results of the command received by the caller of rexec_r() are not converted from CCSID 819. Conversion from ASCII ccsid 819 to the CCSID of the process or thread is the caller's responsibility.
- If a remote authorization failure occurs, the return value will be -1 and if errorDescriptor is non-null a message indicating the authorization failure can be received with the socket descriptor from errorDescriptor.
- Any socket descriptor returned to the caller of rexec_r() must be explicitly closed by the caller.
- The user, password, and command will be translated from the job ccsid to ASCII ccsid 819 to be sent to the remote host.
- Issuing rexec_r() to a remote host that is configured to set up a SOCKSified connection is not supported.

Related Information

- "rexec()—Issue a Command on a Remote Host" on page 132—Issue a Command on a Remote Host

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **rexec_r()** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <qtqiconv.h>
#include <arpa/rexec.h>
#include <errno.h>

#define BufLen 256

void main()
{
  int sd = -1, rc;
  int responseLen = BufLen;
  int outbytesleft = BufLen;
  int bytesRead, saveBytesRead;
  struct hostent_data host_ent_data;
  struct servent serv_ent;
  struct servent_data serv_ent_data;
  char inbuf[BufLen];
  char outbuf[BufLen];
  char *inbufPtr = (char *)inbuf;
  char *outbufPtr = (char *)outbuf;
  iconv_t cd;
  QtqCode_T toCode   = {0,0,0,0,0,0};    /* Convert to job CCSID */
  QtqCode_T fromCode = {819,0,0,1,0,0};  /* ASCII CCSID */
  char *host;
  char remoteHost[256] = "remoteHost";
  char user[32] = "userName";
  char password[32] = "myPassword";
  char cmd[256] = "commandToRun";
  int *errordesc = NULL;

  /* Must zero this out before call or results will be unpredictable. */
  memset(&serv_ent_data.serve_control_blk, 0x00, sizeof(struct netdb_control_block));
```

```
  /* retrieve the rexec server port number */
  rc = getservbyname_r("exec", "tcp", &serv_ent, &serv_ent_data);
  if (rc < 0)
    printf("getservbyname_r() failed with errno = %d\n",errno);

  /* must zero this out before call or results will be unpredictable. */
  memset((void *)&host_ent_data.host_control_blk, 0x00, sizeof(struct netdb_control_block));
  host = remoteHost;
   errno = 0;

  /* issue the rexec_r api */
  sd = rexec_r(&host, serv_ent.s_port, user, password, cmd, errordesc, &host_ent_data);
  if (sd == -1) /* check if rexec_r() failed */
  {
    if (errno)
      printf("rexec_r() failed with errno = %d\n",errno);
    else
      printf("Either the host does not exist or remote authentication failed.\n");
  }
  else /* rexec_r() was successful */
  {
    bytesRead = recv(sd, inbuf, responseLen, 0);
    if (bytesRead > 0)
    {
      saveBytesRead = bytesRead;
      inbuf[bytesRead-1] = 0;  /* Null terminate */
      /* translate from ASCII to EBCDIC */
      cd = QtqIconvOpen(&toCode, &fromCode);
      iconv(cd,
            (unsigned char **)&inbufPtr,
            (unsigned int *)&bytesRead,
            (unsigned char **)&outbufPtr,
            (unsigned int *)&outbytesleft);
      iconv_close(cd);
      outbufPtr -= saveBytesRead;  /* Reset the buffer pointers */
      printf("%s\n",outbufPtr);
    }
    else if (bytesRead == 0)
      printf("The remote host closed the connection.\n");
    else
      printf("recv() failed with errno = %d\n",errno);
  }
  if (sd != -1)
    close(sd);  /* close the connection. */
  return;
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

rexec_r_ts64()—Issue a Command on a Remote Host

Syntax

```
#include <arpa/rexec.h>

int rexec_r_ts64(char * __ptr64 * __ptr64 host,
            int port,
            char *  __ptr64 user,
            char *  __ptr64 password,
            char *  __ptr64 command,
            int *  __ptr64 errorDescriptor,
            struct hostent_data * __ptr64hostEntData);
```

Service Program Name: QSOSRVTS
Default Public Authority: *USE
Threadsafe: Yes

The **rexec_r_ts64()** function is used to open a connection to a remote host and send a user ID, password, and command to the remote host. The remote host verifies that the user ID and password are valid. The command is issued after the user ID and password are validated. **rexec_r_ts64()** differs from **rexec_r()** in that **rexec_r_ts64()** accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, and other related information, see "rexec_r()—Issue a Command on a Remote Host" on page 136.

Usage Notes

All of the usage notes for "rexec_r()—Issue a Command on a Remote Host" on page 136 apply to **rexec_r_ts64()**.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

rexec_ts64()—Issue a Command on a Remote Host

Syntax

```
#include <arpa/rexec.h>

int rexec_ts64(char * __ptr64 * __ptr64 host,
          int port,
          char * __ptr64 user,
          char * __ptr64 password,
          char * __ptr64 command,
          int * __ptr64 errorDescriptor);
```

Service Program Name: QSOSRVTS
Default Public Authority: *USE
Threadsafe: Yes

The **rexec_ts64()** function is used to open a connection to a remote host and send a user ID, password, and command to the remote host. The remote host verifies that the user ID and password are valid. The command is issued after the user ID and password are validated. **rexec_ts64()** differs from **rexec()** in that **rexec_ts64()** accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, and other related information, see "rexec()—Issue a Command on a Remote Host" on page 132.

Usage Notes

All of the usage notes for "rexec()—Issue a Command on a Remote Host" on page 132 apply to **rexec_ts64()**.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

select()—Wait for Events on Multiple Sockets

Syntax

```
#include <sys/types.h>
#include <sys/time.h>

int select(int max_descriptor,
           fd_set *read_set,
           fd_set *write_set,
           fd_set *exception_set,
           struct timeval *wait_time)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 142.

The *select()* function is used to enable an application to multiplex I/O. By using *select()*, an application with multiple interactive I/O sources avoids blocking on one I/O stream while the other stream is ready. Thus, for example, an application that receives inputs from two distinct communication endpoints (using sockets) can use *select()* to sleep until input is available from either of the sources. When input is available, the application wakes up and receives an indication as to which descriptor is ready for reading.

The application identifies descriptors to be checked for read, write, and exception status and specifies a timeout value. If any of the specified descriptors is ready for the specified event (read, write, or exception), *select()* returns, indicating which descriptors are ready. Otherwise, the process waits until one of the specified events occur or the wait times out.

Parameters

max_descriptor
(Input) Descriptors are numbered starting at zero, so the *max_descriptor* parameter must specify a value that is one greater than the largest descriptor number that is to be tested.

read_set
(I/O) A pointer to a set of descriptors that should be checked to see if they are ready for reading. This parameter is a value-result field. Each descriptor to be tested should be added to the set by issuing a FD_SET() macro. If no descriptor is to be tested for reading, *read_set* should be NULL (or point to an empty set). On return from the call, only those descriptors that are ready to be read are in the set. FD_ISSET() should be used to test for membership of a descriptor in the set.

write_set
(I/O) A pointer to a set of descriptors that should be checked to see if they are ready for writing. This parameter is a value-result field. Each descriptor to be tested should be added to the set by issuing a FD_SET() macro. If no descriptor is to be tested for writing, *write_set* should be NULL (or point to an empty set). On return from the call, only those descriptors that are ready to be written are in the set. FD_ISSET() should be used to test for membership of a descriptor in the set.

exception_set
(I/O) A pointer to a set of descriptors that should be checked for pending exception events. This parameter is a value-result field. Each descriptor to be tested should be added to the set by issuing a FD_SET() macro. If no descriptor is to be tested for exceptions, *exception_set* should be NULL (or point to an empty set). On return from the call, only those descriptors that have an exception event are in the set. FD_ISSET() should be used to test for membership of a descriptor in the set.

wait_time
(Input) A pointer to a structure which specifies the maximum time to wait for at least one of the selection criteria to be met. A time to wait of 0 is allowed; this returns immediately with the current status of the sockets. The parameter may be specified even if NO descriptors are specified (*select()* is being used as a timer). If *wait_time* is NULL, *select()* blocks indefinitely. The structure pointed to by the *wait_time* parameter is defined in **<sys/time.h>**.

Authorities

No authorization is required.

Return Value

select() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (if the time limit expires)
- n (total number of descriptors in all sets that met selection criteria)

Note: The **timeval** structure (pointed to by *wait_time*) is unchanged.

Error Conditions

When *select()* fails, *errno* can be set to one of the following:

[EBADF]	Descriptor not valid.
[ENOTSAFE]	Function not allowed.
[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *read_set*, *write_set*, *exception_set*, or *wait_time* parameter.
[EINTR]	Interrupted function call.
[EINVAL]	Parameter not valid. This error code indicates one of the following: • The *max_descriptor* parameter specifies a negative value or a value greater than [FD_SETSIZE]. • The *wait_time* parameter specifies a time value which was not valid.
[EIO]	Input/output error.
[ENOTSUP]	Operation not supported. The operation, though supported in general, is not supported for the requested object or the requested arguments.
[EUNKNOWN]	Unknown system state.

Error Messages

CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred.

Usage Notes

1. » The "poll()—Wait for Events on Multiple Descriptors" on page 78 API is more efficient than the *select()* API and therefore *poll()* is always recommended over *select()*.

   «

2. An application program must include the header file **<sys/types.h>** to use *select()*. The header file contains the type and macro definitions needed to use *select()*. The maximum number of descriptors

that can be selected is defined by FD_SETSIZE. » See **DosSetRelMaxFH()** for additional considerations when *select()* and *DosSetRelMaxFH()* are used within the same process. «

The following macros can be used to manipulate descriptor sets:

- `FD_ZERO`(fd_set *p) removes all descriptors from the set specified by *p*.
- `FD_CLR`(int *n*, fd_set *p) removes descriptor *n* from the set specified by *p*.
- `FD_SET`(int *n*, fd_set *p) adds descriptor *n* to the set specified by *p*.
- `FD_ISSET`(int *n*, fd_set *p) returns a nonzero value if descriptor *n* is returned in the set specified by *p*; otherwise, a zero value is returned.

Note: Values of type fd_set should only be manipulated by the macros supplied in the **<sys/types.h>** header file.

3. A descriptor can be returned in the set specified by *read_set* to indicate one of the following:
   - An error event exists on the descriptor.
   - A connection request is pending on a socket descriptor. This technique can be used to wait for connections on multiple socket descriptors. When a listening socket is returned in the set specified by *read_set*, an application can then issue an *accept()* call to accept the connection.
   - No data can be read from the underlying instance represented by the descriptor. For example, a socket descriptor for which a *shutdown()* call has been done to disable the reception of data.
4. A descriptor can be returned in the set specified by *write_set* to indicate one of the following:
   - Completion of a non-blocking *connect()* call on a socket descriptor. This allows an application to set a socket descriptor to nonblocking (with *fcntl()* or *ioctl()*), issue a *connect()* and receive `[EINPROGRESS]`, and then use *select()* to verify that the connection has completed.
   - No data can be written to the underlying instance represented by the descriptor (for example, a socket descriptor for which a *shutdown()* has been done to disable the sending of data).
   - When a *write()* can be successfully issued without blocking (or, for nonblocking, so it does not return `[EWOULDBLOCK]`).
5. A socket descriptor is returned in the set specified by *exception_set* to indicate that out-of-band data has arrived at the socket. This is only supported for connection-oriented sockets with an address family of `AF_INET` or `AF_INET6`.
6. Unpredictable results will appear if this function or any of its associated type and macro definitions are used in a thread executing one of the scan-related exit programs (or any of its' created threads). See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.

Related Information

- "poll()—Wait for Events on Multiple Descriptors" on page 78—Wait for events on Multiple Sockets
- **DosSetRelMaxFH()**—Change Maximum Number of File Descriptors

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

send()—Send Data

BSD 4.3 Syntax

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

int send(int socket_descriptor,
         char *buffer,
         int buffer_length,
         int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t send(int socket_descriptor,
         const void *buffer,
         size_t buffer_length,
         int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *send()* function is used to send data through a connected socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The socket descriptor that is to be written to.

buffer (Input) The pointer to the buffer in which the data that is to be written is stored.

buffer_length
(Input) The length of the *buffer*.

flags (Input) A flag value that controls the transmission of the data. The *flags* value is either zero, or is obtained by performing an OR operation on the following constants:

MSG_EOR	Terminate a record, if supported by the protocol.
MSG_OOB	Send data as out-of-band data. Valid only for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
MSG_DONTROUTE	Bypass routing. Valid only for sockets with address family of AF_INET. It is ignored for other address families.

Authorities

No authorization is required.

Return Value

send() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes sent.

Error Conditions

When *send()* fails, *errno* can be set to one of the following:

[EACCES] Permission denied.

This error code indicates one of the following:

- Destination address specified a broadcast address and the socket option `SO_BROADCAST` was not set (with a *setsockopt()*).
- The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with a type of `SOCK_DGRAM` and an address family of `AF_INET`.

[EBADF] Descriptor not valid.

[ECONNREFUSED] The destination socket refused an attempted connect operation.

This error code can only be returned on sockets that use a connectionless transport service.

[EDESTADDRREQ] Operation requires destination address.

A destination address has not been associated with the socket pointed to by the *socket_descriptor* parameter. This error code can only be returned on sockets that use a connectionless transport service.

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *buffer* parameter.

[EHOSTDOWN] A remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EHOSTUNREACH] A route to the remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

The *buffer_length* parameter specifies a negative value.

[EIO] Input/output error.

[EMSGSIZE] Message size out of range.

The data to be sent could not be sent atomically because the size specified by *buffer_length* is too large.

[ENETDOWN] The network is not currently available.

This error code can only be returned on sockets that use a connectionless transport service.

[ENETUNREACH] Cannot reach the destination network.

This error code can only be returned on sockets that use a connectionless transport service.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTCONN] Requested operation requires a connection.

This error code can only be returned on sockets that use a connection-oriented transport service.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but the *socket_descriptor* parameter points to a connectionless socket.
- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but the *socket_descriptor* parameter points to a socket that does not have an address family of `AF_INET` or `AF_INET6`.

[EPIPE] Broken pipe.

[EUNATCH]	The protocol required to support the specified address family is not available at this time.
[EUNKNOWN]	Unknown system state.
[EWOULDBLOCK]	Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. *send()* only works with sockets on which a *connect()* has been issued, since it does not allow the caller to specify a destination address.
2. To broadcast on an `AF_INET` socket, the socket option `SO_BROADCAST` must be set (with a *setsockopt()*).
3. When using a connection-oriented transport service, all errors except `[EUNATCH]` and `[EUNKNOWN]` are mapped to `[EPIPE]` on an output operation when either of the following occurs:
   - A connection that is in progress is unsuccessful.
   - An established connection is broken.

   To get the actual error, use *getsockopt()* with the `SO_ERROR` option, or perform an input operation (for example, *read()*).
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *send()* API is mapped to *qso_send98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 68—Perform I/O Control Request
- “sendto()—Send Data” on page 153—Send Data
- “sendmsg()—Send a Message Over a Socket”—Send Data or Descriptors or Both
- “write()—Write to Descriptor” on page 185—Write to Descriptor
- “writev()—Write to Descriptor Using Multiple Buffers” on page 192—Write to Descriptor Using Multiple Buffers

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

sendmsg()—Send a Message Over a Socket

BSD 4.3 Syntax

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

int sendmsg(int socket_descriptor,
            struct msghdr *message_structure,
            int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t sendmsg(int socket_descriptor,
                const struct msghdr *message_structure,
                int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *sendmsg()* function is used to send data or descriptors or ancillary data or a combination of these through a connected or unconnected socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The socket descriptor that is to be written to.

message_structure
(I/O) The pointer to the message structure that contains the following:

- The address to which the message is to be sent
- The vector array in which the data to be sent is stored
- The ancillary data; or an access rights list in which the descriptors to be sent are stored.

The structure pointed to by the *message_structure* parameter is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct msghdr {
  caddr_t       msg_name;
  int           msg_namelen;
  struct iovec *msg_iov;
  int           msg_iovlen;
  caddr_t       msg_accrights;
  int           msg_accrightslen;
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
struct msghdr {
  void         *msg_name;
  socklen_t     msg_namelen;
  struct iovec *msg_iov;
  int           msg_iovlen;
  void         *msg_control;      /* Set to NULL if not needed  */
  socklen_t     msg_controllen;   /* Set to 0 if not needed     */
  int           msg_flags;
};
```

The *msg_name* and *msg_namelen* fields contain the address and address length to which the message is sent. For further information on the structure of socket addresses, see Sockets Programming in the iSeries Information Center. If the *msg_name* field is set to a NULL pointer, the address information is not returned.

The *msg_iov* and *msg_iovlen* fields are for scatter/gather I/O.

The BSD 4.3 structure uses the *msg_accrights* and *msg_accrightslen* fields to pass descriptors. The *msg_accrights* field is a list of zero or more descriptors, and *msg_accrightslen* is the total length (in bytes) of the descriptor list.

The BSD 4.4/UNIX 98 compatible structure uses the *msg_control* and *msg_controllen* fields to pass ancillary data. The *msg_control* field is a pointer to ancillary data (of length *msg_controllen*) with the form:

```
struct cmsghdr {
      socklen_t cmsg_len;    /* # bytes, including this header */
      int       cmsg_level;  /* originating protocol           */
      int       cmsg_type;   /* protocol-specific type         */
                    /* followed by unsigned char cmsg_data[];  */
};
```

The *cmsg_len* field is the total length including this header. *cmsg_level* is the originating protocol. *cmsg_type* is the protocol-specific type. If ancillary data is not being passed, the *msg_control* field must be initalized to NULL and the *msg_controllen* field must be initialized to 0. The following tables list the supported ancillary data types when using the BSD 4.4/UNIX 98 compatible structures.

Ancillary Data Types That Apply to the Socket Layer (where *cmsg_level* is SOL_SOCKET):

cmsg_type	cmsg_data
SCM_RIGHTS	The rest of the buffer is a list of zero or more descriptors to be sent. This ancillary data type is only supported for sockets with an address family of AF_UNIX or AF_UNIX_CCSID.

Ancillary Data Types That Apply to the IP Layer (where *cmsg_level* is IPPROTO_IP):

cmsg_type	cmsg_data
IP_QOS_CLASSIFICATION_DATA	The rest of the buffer is an *ip_qos_classification_data* structure. This structure is defined in **<netinet/ip.h>**. For further information on the how this structure should be initialized, see Quality of Service in the iSeries Information Center. This ancillary data type is only supported for sockets with an address family of AF_INET and a type of SOCK_STREAM.

Macros are provided for navigating these structures.

- *CMSG_DATA(cmsg)* If the argument is a pointer to a *cmsghdr* structure, this macro returns an unsigned character pointer to the data array associated with the *cmsghdr* structure.
- *CMSG_NXTHDR(mhdr,cmsg)* If the first argument is a pointer to a *msghdr* structure and the second argument is a pointer to a *cmsghdr* structure in the ancillary data, pointed to by the msg_control field of that msghdr structure, this macro returns a pointer to the next *cmsghdr* structure, or a null pointer if this structure is the last *cmsghdr* in the ancillary data.
- *CMSG_FIRSTHDR(mhdr)* If the argument is a pointer to a *msghdr* structure, this macro returns a pointer to the first *cmsghdr* structure in the ancillary data associated with this *msghdr* structure, or a null pointer if there is no ancillary data associated with the *msghdr* structure.

The BSD 4.4/UNIX 98 *msg_flags* field is ignored for sendmsg().

flags (Input) A flag value that controls the transmission of the data. The *flags* value is either zero, or is obtained by performing an OR operation on one or more of the following constants:

MSG_EOR	Terminate a record, if supported by the protocol.
MSG_OOB	Send data as out-of-band data. Valid only for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
MSG_DONTROUTE	Bypass routing. Valid only for sockets with address family of AF_INET. It is ignored for other address families.

Authorities

When the address family of the socket identified by the ***socket_descriptor*** is AF_INET and is running IP over SNA, the thread must have retrieve, insert, delete, and update authority to the APPC device. When the thread does not have this level of authority, an *errno* of EACCES is returned.

Return Value

sendmsg() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes sent.

Error Conditions

When *sendmsg()* fails, *errno* can be set to one of the following:

[EACCES]	Permission denied. The process does not have the appropriate privileges to the destination address.
[EADDRNOTAVAIL]	Address not available. A socket with an address family of AF_INET or AF_INET6 is using a connectionless transport service, the socket was not bound. The system tried to bind the socket but could not because a port was not available.
[EBADF]	Descriptor not valid.
[ECONNREFUSED]	The destination socket refused an attempted connect operation. This error code can only be returned on sockets that use a connectionless transport service.
[EDESTADDRREQ]	Operation requires destination address. A destination address has not been associated with the socket pointed to by the *socket_descriptor* parameter and a destination address was not set in the *msghdr* structure (pointed to by the *message_structure* parameter). This error code can only be returned on sockets that use a connectionless transport service.
[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *message_structure* parameter or a field within the structure pointed to by the *message_structure* parameter.
[EHOSTDOWN]	A remote host is not available. This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTUNREACH]	A route to the remote host is not available. This error code can only be returned on sockets that use a connectionless transport service.
[EINTR]	Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *msg_iovlen* field or the **iov_len** field in a **iovec** structure specifies a negative value. The fields are contained in the *msghdr* structure (pointed to by the *message_structure* parameter).
- The *msg_namelen* field in the *msghdr* structure (pointed to by the *message_structure* parameter) specifies a length that is not valid for the address family.
- The *msg_accrightslen* field in the *msghdr* structure specifies a negative value or is not large enough when *msg_accrights* was specified.
- The *msg_controllen* field in the *msghdr* structure specifies a negative value or is not large enough when *msg_control* was specified.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and the CCSID specified in *sunc_qlg* in the **sockaddr_unc** structure (pointed to by *local_address*) cannot be converted to the current default CCSID for integrated file system path names.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and there was an incomplete character or shift state sequence at the end of *sunc_path* in the **sockaddr_unc** structure (pointed to by *local_address*).
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and the **sockaddr_unc** structure (pointed to by *local_address*) was not valid:
  - The sunc_format was not set to SO_UNC_DEFAULT or SO_UNC_USE_QLG.
  - The sunc_zero was not initialized to zeros.
  - The sunc_format field was set to SO_UNC_USE_QLG and the sunc_qlg structure was not valid:
    - The path type was less than 0 or greater than 3.
    - The path length was less than 0 or out of bounds. For example, a single-byte path name was greater than 126 bytes or a double-byte path name was greater than 252 bytes.
    - A reserved field was not initialized to zeros.

[EIO] Input/output error.

[EISCONN] A connection has already been established.

A destination address was set, but the socket pointed to by the *socket_descriptor* parameter already has a destination address associated with it.

[ELOOP] A loop exists in symbolic links encountered during pathname resolution.

This error code refers to the destination address, and can only be returned by sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[EMSGSIZE] Message size out of range.

This error code indicates one of the following:

- The data to be sent could not be sent atomically because the total size of the data to be sent is too large.
- The *msg_iovlen* field in the *msghdr* structure (pointed to by the *message_structure* parameter) specifies a value that is greater than [MSG_MAXIOVLEN] (defined in **<sys/socket.h>**).

[ENAMETOOLONG] File name too long.

This error code refers to the destination address, and can only be returned by sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENETDOWN] The network is not currently available.

This error code can only be returned on sockets that use a connectionless transport service.

[ENETUNREACH] Cannot reach the destination network.

This error code can only be returned on sockets that use a connectionless transport service.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOENT] No such file or directory.

This error code refers to the destination address, and can only be returned by sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOSYS] Function not implemented.

This error code refers to the destination address, and can only be returned by sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOTCONN] Requested operation requires a connection.

This error code can only be returned on sockets that use a connection-oriented transport service.

[ENOTDIR] Not a directory.

This error code refers to the destination address, and can only be returned by sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but the *socket_descriptor* parameter points to a connectionless socket.
- The *flags* parameter specifies a value that includes the `MSG_OOB` flag, but the *socket_descriptor* parameter points to a socket that does not have an address family of `AF_INET` or `AF_INET6`.
- The *msg_accrights* and *msg_accrightslen* (or the BSD 4.4/UNIX 98 compatible fields *msg_control* and *msg_controllen*) were specified and the underlying instance represented by the descriptor does not support the passing of access rights.

[EPIPE] Broken pipe.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The passing of descriptors is only supported over sockets that have an address family of `AF_UNIX` or `AF_UNIX_CCSID`. The *msg_accrightslen* and the *msg_accrights* fields (or the BSD 4.4/UNIX 98 compatible fields *msg_control* and *msg_controllen*) are ignored if the socket has any other address family. When you use *sendmsg()* and *recvmsg()* to pass descriptors, the target job must be running with either of the following:
   - The same user profile as the source job (in essence, passing the descriptor to yourself)
   - *ALLOBJ special authority

If the target job closes the receiving end of the UNIX domain socket while a descriptor is in transit, the descriptor is reclaimed by the system, and the resource that it represented is closed. For files and directories, the ability to pass descriptors using *sendmsg()* and *recvmsg()* is only supported for objects in Root, QOpenSys, User-defined file systems (UDFS), and Network File System (NFS).

2. *sendmsg()* is an atomic operation in that it produces one packet of data each time the call is issued on a connectionless socket. For example, a *sendmsg()* to a datagram socket will result in a single datagram.
3. A destination address cannot be specified if the socket pointed to by the *socket_descriptor* parameter already has a destination address associated with it. To **not** specify an address, users must set the *msg_name* field to NULL or set the *msg_namelen* field to zero. (Not specifying an address by setting the *msg_namelen* field to zero is an IBM extension.)

   Note: The *msg_name* and *msg_namelen* fields are ignored if the socket is using a connection-oriented transport service.
4. If the socket is using a connectionless transport device, the socket is not bound to an address, and the socket type is `SOCK_DGRAM`, the system automatically selects an address (INADDR_ANY or in6addr_any and an available port number) and binds it to the socket before sending the data.
5. To broadcast on an `AF_INET` socket, the socket option `SO_BROADCAST` must be set (with a *setsockopt()*).
6. When using a connection-oriented transport service, all errors except `[EUNATCH]` and `[EUNKNOWN]` are mapped to `[EPIPE]` on an output operation when either of the following occurs:
   - A connection that is in progress is unsuccessful.
   - An established connection is broken.

   To get the actual error, use *getsockopt()* with the `SO_ERROR` option, or perform an input operation (for example, *read()*).
7. If the socket is using an address family of `AF_UNIX`, the destination address (which is a path name) is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job. For `AF_UNIX_CCSID`, the destination address is assumed to be in the format and coded character set identifier (CCSID) specified in the **sockaddr_unc**.
8. For `AF_INET` sockets over SNA, type `SOCK_DGRAM`, if a datagram can not be delivered, no errors are returned. (As an example, a datagram might not be delivered if there is no datagram application at the remote host listening at the requested port.)
9. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *sendmsg()* API is mapped to *qso_sendmsg98()*.

Related Information

- For additional information and sample programs on how to use sendmsg() and recvmsg() to pass descriptors between iSeries jobs, see Socket Programming in the iSeries Information Center.
- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 68—Perform I/O Control Request
- “givedescriptor()—Pass Descriptor Access to Another Job” on page 60—Pass Descriptor Access to Another Job
- “send()—Send Data” on page 143—Send Data
- “sendto()—Send Data” on page 153—Send Data
- “takedescriptor()—Receive Socket Access from Another Job” on page 183—Receive Socket Access from Another Job

API introduced: V3R1

sendto()—Send Data

BSD 4.3 Syntax

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

int sendto(int socket_descriptor,
           char *buffer,
           int buffer_length,
           int flags,
           struct sockaddr *destination_address,
           int address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

ssize_t sendto(int socket_descriptor,
           const void *buffer,
           size_t buffer_length,
           int flags,
           const struct sockaddr *destination_address,
           socklen_t address_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *sendto()* function is used to send data through a connected or unconnected socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The socket descriptor that is to be written to.

buffer (Input) The pointer to the buffer in which the data that is to be written is stored.

buffer_length
(Input) The length of the *buffer*.

flags (Input) A flag value that controls the transmission of the data. The *flags* value is either zero, or is obtained by performing an OR operation on one or more of the following constants:

MSG_EOR	Terminate a record, if supported by the protocol.
MSG_OOB	Send data as out-of-band data. Valid only for sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
MSG_DONTROUTE	Bypass routing. Valid only for sockets with address family of AF_INET. It is ignored for other address families.

destination_address
(Input) A pointer to a buffer of type **struct sockaddr** that contains the destination address to which the data is to be sent. The structure **sockaddr** is defined in **<sys/socket.h>**.

The BSD 4.3 structure is:

```
struct sockaddr {
   u_short sa_family;
   char    sa_data[14];
};
```

The BSD 4.4/UNIX 98 compatible structure is:

```
typedef uchar   sa_family_t;

struct sockaddr {
   uint8_t     sa_len;
   sa_family_t sa_family;
   char        sa_data[14];
};
```

The BSD 4.4 *sa_len* field is the length of the address. The *sa_family* field identifies the address family to which the address belongs, and *sa_data* is the address whose format is dependent on the address family.

address_length
(Input) The length of the *destination_address*.

Authorities

When the address family of the socket identified by the ***socket_descriptor*** is AF_INET and is running IP over SNA, the thread must have retrieve, insert, delete, and update authority to the APPC device. When the thread does not have this level of authority, an *errno* of EACCES is returned.

Return Value

sendto() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the number of bytes sent.

Error Conditions

When *sendto()* fails, *errno* can be set to one of the following:

[EACCES]	Permission denied. The process does not have the appropriate privileges to the destination address.
[EADDRNOTAVAIL]	Address not available. A socket with an address family of AF_INET or AF_INET6, is using a connectionless transport service, and the socket was not bound. The system tried to bind the socket but could not because a port was not available.
[EBADF]	Descriptor not valid.
[ECONNREFUSED]	The destination socket refused an attempted connect operation. This error code can only be returned on sockets that use a connectionless transport service.
[EDESTADDRREQ]	Operation requires destination address. A destination address has not been associated with the socket pointed to by the *socket_descriptor* parameter and a destination address was not passed in as an argument on the *sendto()*. This error code can only be returned on sockets that use a connectionless transport service.
[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *buffer* or *destination_address* parameter.

[EHOSTDOWN] A remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EHOSTUNREACH] A route to the remote host is not available.

This error code can only be returned on sockets that use a connectionless transport service.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *buffer_length* parameter specifies a negative value.
- The socket is using a connectionless transport service and the *address_length* parameter specifies a length that is not valid for the address family.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and the CCSID specified in *sunc_qlg* in the **sockaddr_unc** structure (pointed to by *local_address*) cannot be converted to the current default CCSID for integrated file system path names.
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and there was an incomplete character or shift state sequence at the end of *sunc_path* in the **sockaddr_unc** structure (pointed to by *local_address*).
- The *socket_descriptor* points to a socket with an address family of `AF_UNIX_CCSID`, and the **sockaddr_unc** structure (pointed to by *local_address*) was not valid:
  - The sunc_format was not set to SO_UNC_DEFAULT or SO_UNC_USE_QLG.
  - The sunc_zero was not initialized to zeros.
  - The sunc_format field was set to SO_UNC_USE_QLG and the sunc_qlg structure was not valid:
    - The path type was less than 0 or greater than 3.
    - The path length was less than 0 or out of bounds. For example, a single-byte path name was greater than 126 bytes or a double-byte path name was greater than 252 bytes.
    - A reserved field was not initialized to zeros.

[EIO] Input/output error.

[EISCONN] A connection has already been established.

A destination address was set, but the socket pointed to by the *socket_descriptor* parameter already has a destination address associated with it.

[ELOOP] A loop exists in symbolic links encountered during pathname resolution.

This error code refers to the destination address, and can only be returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[EMSGSIZE] Message size out of range.

The data to be sent could not be sent atomically because the total size of the data to be sent is too large.

[ENAMETOOLONG] File name too long.

This error code refers to the destination address, and can only be returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENETDOWN] The network is not currently available.

This error code can only be returned on sockets that use a connectionless transport service.

[ENETUNREACH] Cannot reach the destination network.

This error code can only be returned on sockets that use a connectionless transport service.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOENT] No such file or directory.

This error code refers to the destination address, and can only be returned on sockets that use the `AF_UNIX` or `AF_UNIX_CCSID` address family.

[ENOSYS] Function not implemented.

This error code refers to the destination address, and can only be returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.

[ENOTCONN] Requested operation requires a connection.

This error code can only be returned on sockets that use a connection-oriented transport service.

[ENOTDIR] Not a directory.

This error code refers to the destination address, and can only be returned on sockets that use the AF_UNIX or AF_UNIX_CCSID address family.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

This error code indicates one of the following:

- The *flags* parameter specifies a value that includes the MSG_OOB flag, but the *socket_descriptor* parameter points to a connectionless socket.
- The *flags* parameter specifies a value that includes the MSG_OOB flag, but the *socket_descriptor* parameter points to a socket that does not have an address family of AF_INET or AF_INET6.

[EPIPE] Broken pipe.

[EPROTOTYPE] The socket type or protocols are not compatible.

This error code is only returned on sockets that use the AF_UNIX or the AF_UNIX_CCSID address family.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

[EWOULDBLOCK] Operation would have caused the thread to be suspended.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. A destination address cannot be specified if the socket pointed to by the *socket_descriptor* parameter already has a destination address associated with it. To **not** specify an address, users must set the *destination_address* field to NULL or set the *address_length* field to zero. (Not specifying an address by setting the *address_length* field to zero is an IBM extension.)

   Note: The *destination_address* and *address_length* fields are ignored if the socket is using a connection-oriented transport service.
2. If the socket is using a connectionless transport device, the socket is not bound to an address, and the socket type is SOCK_DGRAM, the system automatically selects an address (INADDR_ANY or in6addr_any and an available port number) and binds it to the socket before sending the data.
3. To broadcast on an AF_INET socket, the socket option SO_BROADCAST must be set (with a *setsockopt()*).
4. When using a connection-oriented transport service, all errors except [EUNATCH] and [EUNKNOWN] are mapped to [EPIPE] on an output operation when either of the following occurs:
   - A connection that is in progress is unsuccessful.
   - An established connection is broken.

To get the actual error, use *getsockopt()* with the SO_ERROR option, or perform an input operation (for example, *read()*).

5. If the socket is using an address family of AF_UNIX, the destination address (which is a path name) is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job. For AF_UNIX_CCSID, the destination address is assumed to be in the format and coded character set identifier (CCSID) specified in the **sockaddr_unc**.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *sendto()* API is mapped to *qso_sendto98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- "send()—Send Data" on page 143—Send Data
- "sendmsg()—Send a Message Over a Socket" on page 146—Send Data or Descriptors or Both

API introduced: V3R1

---

send_file()—Send a File over a Socket Connection

Syntax

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

int send_file(int *socket_descriptor,
              struct sf_parms *sf_struct,
              int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Conditional; see "**Usage Notes**" on page 160.

The *send_file()* function is used to send the contents of an open file over an existing socket connection.

The *send_file()* API is a combination of the IFS *read()* and the sockets *send()* and *close()* APIs. Socket applications that transmit a file over a socket connection can, under certain circumstances, obtain improved performance by using *send_file()*.

Parameters

socket_descriptor
(Input/Output) A pointer to the socket descriptor that is to be written to.

sf_struct
(Input/Output) A pointer to the send_file structure that contains the following:

- The header buffer and length
- The file descriptor, the offset into the file, the file size, and number of bytes to send from the file
- The trailer buffer and length
- The number of bytes of data that were sent

The structure pointed to by the *sf_struct* parameter is defined in **<sys/socket.h>**.

```
struct sf_parms
{
   void    *header_data;
   size_t   header_length;

   int      file_descriptor;
   size_t   file_size;
   off_t    file_offset;
   ssize_t  file_bytes;

   void    *trailer_data;
   size_t   trailer_length;

   size_t   bytes_sent;
}
```

header_data (Input/Output) A pointer to a buffer that contains data to be sent before the file data is sent.

header_length (Input/Output) The length in bytes of *header_data*.

file_descriptor (Input) The file descriptor for a file that has been opened for read access. This is the descriptor for the file that contains the data to be transmitted. This field is ignored if the *file_bytes* field is set to 0.

file_size (Output) The size in bytes of the file associated with *file_descriptor*.

file_offset (Input/Output) The byte offset into the file from which to start sending data. Specify a value of 0 to start sending data from the start of the file. If a negative value is passed in, *send_file()* API will return with -1 and the *errno* will be set to EINVAL.

file_bytes (Input/Output) The number of bytes from the file to be transmitted. Set the *file_bytes* field to -1 to transmit all of the data from the *file_offset* position in the file to the end of the file. If the *file_bytes* field is set to 0, no data from the file will be transmitted.

trailer_data (Input/Output) A pointer to a buffer that contains data to be sent after the file data is sent.

trailer_length (Input/Output) The length in bytes of *trailer_data*.

bytes_sent (Output) The number of bytes that have been successfully sent.

flags (Input) A flag value that controls what is done with the socket connection after the data has been transmitted. The *flags* value is either zero or it is one of the following constants:

SF_CLOSE After the *header_data*, file data, and *trailer_data* have been successfully sent, the connection and the socket descriptor are closed. The descriptor that is pointed to by the *socket_descriptor* parameter is set to -1 before the *send_file()* API returns to the application.

SF_REUSE After the *header_data*, file data, and *trailer_data* have been successfully sent, the connection is closed. If socket reuse is supported, the descriptor that is pointed to by the *socket_descriptor* parameter is reset. If socket reuse is not supported, the descriptor that is pointed to by the *socket_descriptor* parameter is closed and set to -1.

Authorities

No authorization is required.

Return Value

send_file() returns an integer. Possible values are:

- -1 (unsuccessful call) Check *errno* for additional information
- 0 (successful call) All of the data has been successfully sent
- 1 (interrupted call) The command was interrupted while sending data

Error Conditions

When *send_file()* fails, *errno* can be set to one of the following:

[EACCES] Permission denied.

An attempt was made to access an object in a way forbidden by its object access permissions. A thread does not have access to the specified file, directory, component, or path.

If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System takes place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems.

[EBADF] Descriptor not valid.

This error code indicates one of the following:

- The descriptor pointed to by the *socket_descriptor* parameter is not a valid socket descriptor.
- The *file_descriptor* parameter is not valid for this operation. The specified descriptor is incorrect, does not refer to an open file, or refers to a file that was only open for writing.

[ECONVERT] Conversion error.

[EFAULT] Bad address.

The system detected an address that was not valid while attempting to access the *socket_descriptor* or one of the fields in the send_file structure.

[EINTR] Interrupted function call.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- A NULL pointer was specified for the *sf_struct* parameter
- The *file_offset* parameter specified a negative value.
- The *file_offset* parameter specified a value that was greater than the file size.
- The *file_bytes* parameter would have resulted in a read operation beyond the end of the file.
- The *flags* parameter specified a value that was not valid.

[EIO] Input/output error.

[ENOBUFS] There is not enough buffer space for the requested operation.

[ENOTCONN] Requested operation requires a connection.

[ENOTSAFE] Function is not allowed in a job that is running with multiple threads.

[ENOTSOCK] The specified descriptor does not reference a socket.

[EOPNOTSUPP] Operation not supported.

The *socket_descriptor* parameter references a socket that does not support the *send_file()* function. The *send_file()* function is only valid on sockets that have an address family of `AF_INET`, `AF_INET6`, `AF_UNIX`, or `AF_UNIX_CCSID` and a socket type of `SOCK_STREAM`.

[EOVERFLOW] Object is too large to process.

This error code indicates one of the following:

- The size of the file associated with *file_descriptor* parameter is greater than 2 GB minus 1 byte.
- The total number of bytes to be sent, *header_length* + *file_bytes* + *trailer_length*, is greater than 4 GB minus 1, the largest value that can be stored in the *bytes_sent* output field.

[EPIPE] Broken pipe.

[EUNATCH] The protocol required to support the specified address family is not available at this time.

[EUNKNOWN] Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.

Message ID	Error Message Text
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred.

Usage Notes

1. The *send_file()* function is only valid on sockets that have an address family of `AF_INET`, `AF_INET6`, `AF_UNIX`, or `AF_UNIX_CCSID` and a socket type of `SOCK_STREAM`. If the descriptor pointed to by the *socket_descriptor* parameter does not have the correct address family and socket type, -1 is returned and the *errno* value is set to EOPNOTSUPP.
2. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:
     - Root
     - QOpenSys
     - User-defined
     - QNTC
     - QSYS.LIB
     - QOPT
     - Network File System
     - QFileSvr.400
3. The *file_offset* parameter is used to specify a base zero location in the file referenced by the *file_descriptor* parameter. If the *file_bytes* parameter is set to a value of 1 and the *file_offset* parameter is set to a value of 0, the first byte from the file is sent. If the *file_offset* parameter is set to a value of 1, the second byte from the file is sent.
4. An application that uses the *send_file()* API may specify the O_SHARE_RDONLY or the O_SHARE_NONE option on the *open()* call when the file represented by *file_descriptor* is first opened. These options prevent other jobs or threads on the system from updating the file while it is being transmitted.
5. If the O_TEXTDATA option was specified on the *open()* call when the file represented by *file_descriptor* was first opened, the data is sent from the file assuming it is in textual form. The data is converted from the code page of the file to the code page of the application, job, or system as follows:
   - When reading from a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
   - When reading from record files that are being used as stream files, end-of-line characters are added to the end of the data in each record.

   If O_TEXTDATA was not specified on the *open()* call, the data is sent from the file without conversion.

   Regardless of whether or not O_TEXTDATA was specified on the *open()* call, the *header_data* and *trailer_data* are not translated. It is the application's responsibility to translate the *header_data* and *trailer_data* to the correct code page before calling *send_file()*. The *send_file()* function will not translate the data buffers pointed to by the *header_data* and *trailer_data* parameters prior to sending them.

   Note: The ability to do code-page translation is an i5/OS specific extension to the *send_file()* API. The overhead to translate the file will have an effect on the performance of the *send_file()* API.
6. The *send_file()* function attempts to write *header_length* from the buffer pointed to by *header_data*, followed by *file_bytes* from the file associated with *file_descriptor*, followed by *trailer_length* from the buffer pointed to by *trailer_data*, over the connection associated with *socket_descriptor*. As the data is

sent, the API will update the variables in the sf_parms structure so that if the *send_file()* API is interrupted by a signal, the application simply needs to reissue the *send_file()* call using the same parameters.

Note: The value that is passed in for the *flags* parameter is ignored if the *send_file()* API is interrupted by a signal.

7. When you develop in C-based languages and this function is compiled with _LARGE_FILES defined, it will be mapped to *send_file64()*. Note that the type of the *sf_struct* parameter, `struct sf_parms *`, also will be mapped to type `struct sf_parms64 *`.

Related Information

- "accept_and_recv()—Wait for Connection Request and Receive the First Message That Was Sent" on page 8—Wait for Connection Request and Receive the First Message That Was Sent
- "close()—Close File or Socket Descriptor" on page 19—Close File or Socket Descriptor
- open()—Open File
- "send()—Send Data" on page 143—Send Data

API introduced: V4R3

Top | UNIX-Type APIs | APIs by category

---

send_file64()—Send a File over a Socket Connection

Syntax

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

int send_file64(int *socket_descriptor,
                struct sf_parms64 *sf_struct,
                int flags)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 162.

The *send_file64()* function is used to send the contents of an open file over an existing socket connection.

The *send_file64()* API is a combination of the IFS *read()* and the sockets *send()* and *close()* APIs. Socket applications that transmit a file over a socket connection can, under certain circumstances, obtain improved performance by using *send_file64()*.

send_file64() is enabled for large files. It is capable of operating on files larger than 2 GB minus 1 byte. For additional information on the parameters, authorities required, return values, error conditions, error messages, and other usage notes, see "send()—Send Data" on page 143.

Parameters

socket_descriptor
(Input/Output) A pointer to the socket descriptor that is to be written to.

sf_struct
(Input/Output) A pointer to the send_file64 structure that contains the following:

- The header buffer and length.

- The file descriptor, the offset into the file, the file size, and the number of bytes to send from the file.
- The trailer buffer and length.
- The number of bytes of data that were sent.

The structure pointed to by the *sf_struct* parameter is defined in **<sys/socket.h>**.

```
struct sf_parms64
{
   void                *header_data;
   size_t               header_length;

   int                  file_descriptor;
   unsigned long long   file_size;
   long long            file_offset;
   long long            file_bytes;

   void                *trailer_data;
   size_t               trailer_length;

   unsigned long long   bytes_sent;
}
```

flags (Input) A flag value that controls what is done with the socket connection after the data has been transmitted.

Authorities

No authorization is required.

Usage Notes

1. When you develop in C-based languages, the prototypes for the 64-bit APIs are normally hidden. To use the *send_file64()* API, you must compile the source with the _LARGE_FILE_API macro defined.
2. All of the Usage Notes for *send_file()* apply to *send_file64()*. See Usage Notes in the *send_file()* API.

API introduced: V4R4

Top | UNIX-Type APIs | APIs by category

---

setdomainname()—Set Domain Name

Syntax

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

int setdomainname(char *name,
                  int length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *setdomainname()* function is used to set the name of the domain.

Parameters

name (Input) The pointer to a character array where the domain name is stored.

length (Input) The length of the *name* parameter. The length can be from 0 to 255 bytes.

Authorities

No authorization is required.

Return Value

setdomainname() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *setdomainname()* fails, *errno* can be set to one of the following:

[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *name* parameter.
[EINVAL]	Parameter not valid. This error code indicates one of the following: • The *length* parameter specifies a negative value or a value that is greater than the allowed maximum length. • The domain name pointed to by the *name* parameter contains characters that do not belong to the invariant character set.
[EIO]	Input/output error.
[EPERM]	Operation not permitted. The process does not have the appropriate privileges to use *setdomainname()*.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. A process must have the `*iosyscfg` special authority to use *setdomainname()*.
2. The name of the domain is set to NULL when the pointer to the domain name (pointed to by the *name* parameter) is set to NULL.
3. *setdomainname()* only allows domain names that are made up of invariant characters. In addition, the domain name is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job.

   Note: For exceptions to the invariant character set for some CCSIDs, see globalization topic.

Related Information

- "getdomainname()—Retrieve Domain Name" on page 42—Retrieve Domain Name

API introduced: V3R1

sethostid()—Set Host ID

Syntax

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

int sethostid(int host_id)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *sethostid()* function is used to set a host ID.

Parameters

host_id
(Input) The 32-bit *host_id*

Authorities

No authorization is required.

Return Value

sethostid() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *sethostid()* fails, *errno* can be set to one of the following:

[EIO]	Input/output error.
[EPERM]	Operation not permitted. The process does not have the appropriate privileges to use *sethostid()*.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. A process must have the `*iosyscfg` special authority to use the *sethostid()*.
2. When a process issues a *sethostid()*, the *host_id* can be accessed by ANY process that issues a *gethostid()*.

3. While many socket implementations refer to the *host_id* as the IP address of the machine, this is not necessarily the case. Many machines that support the TCP/IP protocol suite support multiple local IP addresses. The value contained in *host_id* is **not** used by TCP in any manner.
4. The *host_id* is reset to zero when an initial program load is performed.
5. The *host_id* is a signed integer. Therefore, a user should be careful to not confuse a return value of -1 from a *gethostid()* with an error return value. *gethostid()* never returns an error.

Related Information

"gethostid()—Retrieve Host ID" on page 43—Retrieve Host ID Address

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

sethostname()—Set Host Name

Syntax

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

int sethostname(char *name,
                int length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *sethostname()* function is used to set the name of the host for a system.

Parameters

name (Input) The pointer to a character array where the host name is stored.

length (Input) The length of the *name* parameter.

Authorities

No authorization is required.

Return Value

sethostname() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *sethostname()* fails, *errno* can be set to one of the following:

[EFAULT] Bad address.

The system detected an address which was not valid while attempting to access the *name* parameter.

[EINVAL] Parameter not valid.

This error code indicates one of the following:

- The *length* parameter specifies a negative value or a value that is greater than the allowed maximum length.
- The host name pointed to by the *name* parameter contains characters that are not invariant.

[EPERM] Operation not permitted.

The process does not have the appropriate privileges to use *sethostname()*.

[EIO] Input/output error.

[EUNKNOWN] Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. A process must have the `*iosyscfg` special authority to use the *sethostname()*.
2. Maximum length of host names is defined by [MAXHOSTNAMELEN] (defined in **<sys/param.h>**).
3. The host name can be set in the following two ways (and users should be aware of the implications of the way they choose):
   - By using option 12 (Change local domain and host names) on the Configure TCP/IP (CFGTCP) menu. When option 12 is used to change the local domain name or local host name, the system appends the local domain name to the local host name and stores the value for access by *sethostname()* and *gethostname()*.
   - By using the *sethostname()* function. When *sethostname()* is used to set the host name, the TCP/IP configuration file is not affected. Only the field that is accessed by *sethostname()* and *gethostname()* is changed.
4. The name of the host is set to NULL when the pointer to the host name (pointed to by the *name* parameter) is set to NULL.
5. The host name is assumed to be in the default coded character set identifier (CCSID) currently in effect for the job. In addition, the host name must adhere to the following conventions.
   - The first character must be either an English alphabetic character or a numeric character.
   - The last character must be either an English alphabetic character, a numeric character, or a period (.).
   - Blanks are not allowed (trailing blanks are removed).
   - The special characters period(.), underscore(_), and minus(-) are allowed.
   - Parts of the name separated by periods (.) cannot exceed 63 characters in length.

     Note: Each part of the name separated by periods must begin and end with an English alphanumeric character.
   - Internet address names (in the form nnn.nnn.nnn.nnn (where nnn is a decimal number)) are not allowed.
   - Names must be from 1 to 255 characters in length.

Related Information

"gethostname()—Retrieve Host Name" on page 44—Retrieve Host Name

API introduced: V3R1

---

setsockopt()—Set Socket Options

BSD 4.3 Syntax

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

int setsockopt(int socket_descriptor,
               int level,
               int option_name,
               char *option_value,
               int option_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int setsockopt(int socket_descriptor,
               int level,
               int option_name,
               const void *option_value,
               socklen_t option_length)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *setsockopt()* function is used to set socket options.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket for which options are to be set.

level (Input) Whether the request applies to the socket itself or the underlying protocol being used. Supported values are:

IPPROTO_IP	Request applies to IP protocol layer.
IPPROTO_TCP	Request applies to TCP protocol layer.
SOL_SOCKET	Request applies to socket layer.
IPPROTO_IPV6	Request applies to IPv6 protocol layer.
IPPROTO_ICMPV6	Request applies to ICMPv6 protocol layer.

option_name

(Input) The name of the option to be set. The following tables list the options supported for each *level*. Assume that the option is supported for all address families unless the option is described otherwise.

Note: Options directed to a specific protocol level are only supported by that protocol. An option that is directed to the SOL_SOCKET level always completes successfully. This provides compatibility with **Berkeley Software Distributions** implementations that also shield the application from protocols that do not support an option.

Socket Options That Apply to the IP Layer (IPPROTO_IP)

Option	Description
IP_OPTIONS	Set IP header options. This is only supported for sockets with an address family of AF_INET.
IP_TOS	Set Type Of Service (TOS) and Precedence in the IP header. This option is only supported for sockets with an address family of AF_INET.
IP_TTL	Set Time To Live (TTL) in the IP header. This option is only supported for sockets with an address family AF_INET.
IP_MULTICAST_IF	Set interface over which outgoing multicast datagrams should be sent. An *option_value* parameter of type in_addr is used to specify the local IP address that is associated with the interface over which outgoing multicast datagrams should be sent. An address of INADDR_ANY removes the previous selection. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_MULTICAST_TTL	Set Time To Live (TTL) in the IP header for outgoing multicast datagrams. An *option_value* parameter of type char is used to set this value between 0 and 255. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_MULTICAST_LOOP	Specify that a copy of an outgoing multicast datagram should be delivered to the sending host as long as it is a member of the multicast group. If this option is not set, a copy of the datagram will not be delivered to the sending host. An *option_value* parameter of type char is used to control loopback being on or off. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_ADD_MEMBERSHIP	Joins a multicast group as specified in the *option_value* parameter of type struct ip_mreq. A maximum of IP_MAX_MEMBERSHIPS groups may be joined per socket. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_DROP_MEMBERSHIP	Leaves a multicast group as specified in the *option_value* parameter of type struct ip_mreq. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW.
IP_RECVLCLIFADDR	Indicates if the local interface that a datagram to be received should be returned. A value of 1 indicates the first 4 bytes of the reserved field of the sockaddr structure will contain the local interface. This option is only supported for sockets with an address family of AF_INET and type of SOCK_DGRAM.
IP_DONTFRAG	Set or reset the don't fragment flag in the IP header. This option is supported for sockets with an address family of AF_INET and type of SOCK_DGRAM or SOCK_RAW only.

Socket Options That Apply to the TCP Layer (IPPROTO_TCP)

Option	Description
TCP_NODELAY	Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default, TCP will follow the Nagle algorithm. To disable this behavior, applications can enable TCP_NODELAY to force TCP to always send data immediately. For example, TCP_NODELAY should be used when there is an application using TCP for a request/response. This option is only supported for sockets with an address family of AF_INET or AF_INET6 and type of SOCK_STREAM.

Socket Options That Apply to the Socket Layer (SOL_SOCKET)

Option	Description
SO_ACCEPTECONNABORTED	Enable the listening socket such that a blocking *accept()* will return ECONNABORTED when a connection on the listening backlog is reset prior to the *accept()*. A backlog entry is created when a new connection is established to the listener socket. The *accept()* call will consume all leading invalid entries in the backlog, until a valid entry is located or the backlog is exhausted. If the backlog contains no entries or a valid entry is located, the option has no effect. If the backlog contains at least one invalid entry and there are no valid entries, the *accept()* will return -1 with errno set to ECONNABORTED. The option is only valid on a socket that has successfully issued the *listen()* call. The option has no effect on non-blocking sockets. This option is only used by sockets with an address family of AF_INET or AF_INET6.
SO_BROADCAST	Enable the socket for issuing messages to a broadcast address. This option is only supported for sockets with an address family of AF_INET and type SOCK_DGRAM or SOCK_RAW. The broadcast address to be used may be determined by issuing an *ioctl()* with the SIOCGIFBRDADDR request.
SO_DEBUG	Indicates if low level-debugging is active.
SO_DONTROUTE	Bypass normal routing mechanisms. This option is only supported by sockets with an address family of AF_INET or AF_INET6.
SO_KEEPALIVE	Keep the connection up by sending periodic transmissions. This option is only supported for sockets of an address family of AF_INET or AF_INET6 and type SOCK_STREAM.
SO_LINGER	Indicates if the system attempts delivery of any buffered data or if the system discards it when a *close()* is issued. For sockets that are using a connection-oriented transport service with an address family of AF_INET or AF_INET6, the default is off (which means that the system attempts to send any queued data, with an infinite wait-time).
SO_OOBINLINE	Indicates whether out-of-band data is received inline with normal data. This option is only supported for sockets with an address family of AF_INET or AF_INET6.
SO_RCVBUF	Set the size of the receive buffer.
SO_RCVLOWAT	Set the size of the receive low-water mark. The default size is 1. This option is only supported for sockets with a type of SOCK_STREAM.
SO_RCVTIMEO	Set the receive timeout value. This option is not supported unless _XOPEN_SOURCE is defined to be 520 or greater.
SO_REUSEADDR	Indicates if the local socket address can be reused. This option is supported by sockets with an address family of AF_INET or AF_INET6 and a type of SOCK_STREAM or SOCK_DGRAM.
SO_SNDBUF	Set the size of the send buffer.
SO_SNDLOWAT	Set the size of the send low-water mark. This option is not supported.

Option	Description
SO_SNDTIMEO	Set the send timeout value. This option is not supported unless _XOPEN_SOURCE is defined to be 520 or greater.
SO_USELOOPBACK	Indicates if the loopback feature is being used. This option is not supported.

Socket Options That Apply to the IPv6 Layer (IPPROTO_IPV6)

Option	Description
IPV6_UNICAST_HOPS	Set the hop limit value that will be used for subsequent unicast packets sent by this socket. An *option_value* parameter of type int is used to set this value between 0 and 255. This option is supported for sockets with an address family of AF_INET6 only.
IPV6_MULTICAST_IF	Set the interface over which outgoing multicast datagrams will be sent. An *option_value* parameter of type unsigned int is used to set the interface index that is associated with the interface over which outgoing multicast datagrams will be sent. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_MULTICAST_HOPS	Set the hop limit value that will be used for subsequent multicast packets sent by this socket. An *option_value* parameter of type int is used to set this value between 0 and 255. If IPV6_MULTICAST_HOPS is not set, the default is 1. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_MULTICAST_LOOP	Set the multicast looping mode. A value of 1 (default), indicates that multicast datagrams sent by this system should also be delivered to this system as long as it is a member of the multicast group. If this option is 0, a copy of the datagram will not be delivered to the sending host. An *option_value* parameter of type unsigned int is used to set this value. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_JOIN_GROUP	Joins a multicast group as specified in the *option_value* parameter of type struct ipv6_mreq. A maximum of IP_MAX_MEMBERSHIPS groups may be joined per socket. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_LEAVE_GROUP	Leaves a multicast group as specified in the *option_value* parameter of type struct ipv6_mreq. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW.
IPV6_V6ONLY	Set the AF_INET6 communication restrictions. A non-zero value indicates that this AF_INET6 socket is restricted to IPv6 communications only. This option stores an int value. This is a boolean option. By default this option is turned off. This option is supported for sockets with an address family of AF_INET6 only.
IPV6_CHECKSUM	Set if the kernel will calculate and insert a checksum for output and verify the received checksum on input, discarding the packet if the checksum is in error for this socket. An *option_value* parameter of type int is used to set this value. If this option is -1 (the default), this socket option is disabled. A value of 0 or greater specifies an integer offset into the user data of where the checksum is located. This must be an even integer value. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_RAW with a protocol other than IPPROTO_ICMPV6. The checksum is automatically computed for protocol IPPROTO_ICMPV6.
IPV6_DONTFRAG	Set if the kernel will not implement the automatic insertion of a fragment header in the packet if the packet is too big for the path MTU. By default this socket option is disabled. Setting the value to 0 also disables the option. If this option is set to a non-zero value the kernel will discard the packet instead of inserting the fragment header. This option is supported for sockets with an address family of AF_INET6 and type of SOCK_DGRAM or SOCK_RAW only.

Socket Options That Apply to the ICMPv6 Layer (IPPROTO_ICMPV6)

Option	Description
ICMP6_FILTER	Set the ICMPv6 Type Filtering. An *option_value* parameter of type **struct icmp6_filter**, defined in **<netinet/icmp6.h>** is used to set this value. The following macros, defined in **<netinet/icmp6.h>** can be used to update the type filtering structure to specify whether or not specific ICMPv6 message types will be passed to the application or be blocked: ICMP6_FILTER_SETPASS, ICMP6_FILTER_SETBLOCK, ICMP6_FILTER_SETPASSALL, and ICMP6_FILTER_SETBLOCKALL. The default is to pass all ICMPv6 message types to the application. This option is only supported for sockets with an address family of AF_INET6 and type of SOCK_RAW with a protocol of IPPROTO_ICMPV6.

option_value

(Input) A pointer to the option value. Integer flags/values are required by *setsockopt()* for all the socket options except SO_LINGER, IP_OPTIONS, IP_MULTICAST_IF, IP_MULTICAST_TTL, IP_MULTICAST_LOOP, IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP, IPV6_JOIN_GROUP, IPV6_LEAVE_GROUP, ICMP6_FILTER.

Note: For the IP_TOS and IP_TTL options, only the rightmost octet (least significant octet) of the integer value is used.

The following options can be set by specifying a nonzero value for the *option_value* parameter:

- <img src="delta.gif" alt="Start of change"> SO_ACCEPTECONNABORTED<img src="deltaend.gif" alt="End of change">
- SO_BROADCAST
- SO_DEBUG
- SO_DONTROUTE
- SO_KEEPALIVE
- SO_OOBINLINE
- SO_REUSEADDR
- TCP_NODELAY
- IP_MULTICAST_LOOP
- IP_DONTFRAG
- IPV6_V6ONLY
- IPV6_MULTICAST_LOOP
- » IPV6_DONTFRAG«

For the SO_LINGER option, *option_value* is a pointer to the structure **struct linger**, defined in **<sys/socket.h>**.

```
struct linger [
  int        l_onoff;
  int        l_linger;
];
```

The *l_onoff* field determines if the linger option is set. A nonzero value indicates the linger option is set and is using the *l_linger* value. A zero value indicates that the option is not set. The *l_linger* field is the time to wait before any buffered data to be sent is discarded. The following occur on a *close()*:

- For AF_INET and AF_INET6 sockets:
  - If the *l_onoff* value is zero, the system attempts to send any buffered data with an infinite wait-time.

– If the *l_onoff* value is nonzero and the *l_linger* value is nonzero, the system attempts to send any buffered data for *l_linger* time. If *l_linger* time has elapsed and the data is still not successfully sent, it is discarded. When data is discarded, the remote program may receive a [ECONNRESET].

- For AF_INET sockets over SNA:

  – If the *l_onoff* value is nonzero and the *l_linger* value is zero, the system waits indefinitely (no timer is implemented). Otherwise, if the *l_onoff* value is nonzero and the *l_linger* value is zero, the system discards any buffered data. When data is discarded, the remote program may receive a [ECONNRESET].

Note: An application must implement an application level confirmation. Guaranteed receipt of data by the partner program is required. Setting SO_LINGER does not guarantee delivery.

For the SO_RCVTIME and SO_SNDTIME options, *option_value* is a pointer to where the structure **timeval** is stored. The structure **timeval** is defined in **<sys/time.h>**.

```
struct timeval {
   long  tv_sec;
   long  tv_usec;
};
```

For the IP_OPTIONS option, *option_value* is a pointer to a character string representing the IP options as specified in RFC 791. The character string varies depending on which options are selected. Each option is made up of a single byte representing the option code, and may be followed by a length field (1 byte) and data for the option. The IP options that can be set are:

- End of option list. Used if options do not end at end of header.
- No operation (used to align octets in a list of options).
- Security and handling restrictions.
- Loose source routing. Used to route a datagram along a path of specified IP addresses. Multiple network hops are allowed between any two IP addresses on the path.
- Record route. Used to trace a route.
- Stream identifier. Used to carry a SATNET stream identifier. This option has been deprecated by RFC 1122 and will result in an error of [EINVAL] if used.
- Strict source routing. Used to route datagram along a path of specified IP addresses. No additional network hops are allowed between any two IP addresses in the path.
- Internet timestamp. Used to record timestamps along the route.

For the IP_MULTICAST_IF option, *option_value* is a pointer to the structure **in_addr**, defined in **<netinet/in.h>** as:

```
struct in_addr [
   u_long  s_addr;
       /* IP address                  */
 ];
```

The **s_addr** field specifies the local IP address that is associated with the interface over which outgoing multicast datagrams should be sent.

For the IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP options, *option_value* is a pointer to the structure **ip_mreq**, defined in **<netinet/in.h>** as:

```
struct ip_mreq [
   struct in_addr imr_multiaddr;          /* IP multicast address of group */
   struct in_addr imr_interface;          /* local IP address of interface */
];
```

The *imr_multiaddr* field is used to specify the multicast group to join or leave. The *imr_interface* field is used to specify the local IP address that is associated with the interface to which this request applies. If INADDR_ANY is specified for the local interface, the default multicast interface will be selected.

» For the IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP options, *option_value* is a pointer to the structure **ipv6_mreq**, defined in **<netinet/in.h>** as:

```
struct ipv6_mreq [
   struct in6_addr ipv6mr_multiaddr;      /* IPv6 multicast address        */
   unsigned int    ipv6mr_interface;      /* interface index               */
];
```

The *ipv6mr_multiaddr* field is used to specify the multicast group to join or leave. The *ipv6mr_interface* field is used to specify the interface to which this request applies. If 0 is specified for the interface, the system will choose the local interface. «

Note: Reception of IP multicast datagrams may require configuration changes to the line description to enable the adapter to receive packets with a multicast destination address. On Ethernet, for example, the Ethernet group address that is associated with the IP group address must be specified by the GRPADR parameter on the line description. To determine the Ethernet group address for a particular IP group address, the low-order 23 bits of the IP address are placed into the low-order 23 bits of the Ethernet group address 01.00.5E.xx.xx.xx.

Notes:

1. For sockets that use a connection-oriented transport service, IP options that are set using *setsockopt()* are only used if they are set prior to a *connect()* being issued. After the connection is established, any IP options that the user sets are ignored.
2. If the IP options portion contains a source routing option, then the address in the source routing option overrides the destination address. The destination address may have been specified on an output operation (for example, on a *sendto()*) or on a *connect()*.
3. If a socket has a type of SOCK_RAW and a protocol of IPPROTO_RAW, any IP options set using *setsockopt()* are ignored (since the user must supply the IP header data on an output operation as part of the data that is being transmitted).

option_length
(Input) The length of the *option_value*.

Authorities

The user profile for the thread must have *IOSYSCFG special authority to set options when the *level* parameter specifies IPPROTO_IP and the *option_value* parameter is IP_OPTIONS.

Return Value

setsockopt() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *setsockopt()* fails, *errno* can be set to one of the following:

[EADDRINUSE]

Address already in use. This error code indicates that the *socket_descriptor* parameter specified for the IP_ADD_MEMBERSHIP operation is already a member of the specified multicast group.

[EADDRNOTAVAIL]

Address not available. For the IP_ADD_MEMBERSHIP or IP_DROP_MEMBERSHIP operations, this error code indicates that an incorrect address was specified for either the *imr_multiaddr* or *imr_interface* parameter value.

[EBADF]

Descriptor not valid.

[ECONNABORTED]

Connection ended abnormally.

This error code indicates that the transport provider ended the connection abnormally because of one of the following:

- The retransmission limit has been reached for data that was being sent on the socket.
- A protocol error was detected.

[EFAULT]

Bad address.

The system detected an address which was not valid while attempting to access the *option_value* parameter.

[EINVAL]

Parameter not valid.

This error code indicates one of the following:

- The *level* parameter specifies a level that is not supported.
- The *option_name* parameter specifies a value that is not valid (except for when the level is SOL_SOCKET , in which case [ENOPROTOOPT] is returned).
- The *option_value* parameter specifies a value that is not valid.
- The *option_length* parameter specifies a negative or zero value.
- An attempt was made to set a socket option that was read-only.

[EIO]

Input/output error.

[ENOBUFS]

There is not enough buffer space for the requested operation.

[ENOPROTOOPT]

The protocol does not support the specified option.

This error code indicates one of the following:

- The socket has an address family of AF_UNIX and the *level* parameter specified is not SOL_SOCKET .
- The *level* parameter specifies a level of SOL_SOCKET and the *option_name* parameter specifies a value that is not valid.

[ENOTCONN]

Requested operation requires a connection.

This error code is only returned if the *level* parameter specifies a level other than SOL_SOCKET and the *socket_descriptor* parameter points to a socket that is using a connection-oriented transport service that has had its connection broken.

[ENOTSOCK]

The specified descriptor does not reference a socket.

[EPERM]

Operation not permitted.

The executing user profile must have *IOSYSCFG special authority to set options when the *level* parameter specifies IPPROTO_IP and the *option_value* parameter is IP_OPTIONS.

[ETOOMANYREFS]

The operation would have exceeded the maximum number of references allowed for this socket.

[EUNATCH]

The protocol required to support the specified address family is not available at this time.

[EUNKNOWN]

Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. Socket options are defined in **<sys/socket.h>**, IP options are defined in **<netinet/ip.h>** and **<netinet/in.h>**, TCP options are defined in **<netinet/tcp.h>**, IPv6 and ICMPv6 options are defined in **<netinet/in.h>**.
2. The following comments applies to the SO_SNDBUF option value:
   - For AF_INET and AF_INET6 sockets over TCP of type SOCK_STREAM, the maximum value the SO_SNDBUF option can be set to is 8 megabytes. Anything greater results in an error of [ENOBUFS]. If the SO_SNDBUF option value is set to a positive value that is less than 512 bytes, the system automatically uses 512 bytes as the SO_SNDBUF size.
   - For AF_INET and AF_INET6 sockets over UDP of type SOCK_DGRAM, the maximum value the SO_SNDBUF option can be set to is 65535 bytes less the IP and UDP header sizes. Anything greater results in an error of [EINVAL].
3. For AF_INET sockets over SNA of type SOCK_STREAM, SO_RCVBUF should be set before connection is established. After connection is established, any changes are ignored. Also, only the client can affect the receive buffer size. The server cannot affect it.
4. For AF_INET sockets over SNA of type SOCK_DGRAM, both SO_SNDBUF and SO_RCVBUF are ignored and have no effect on processing.
5. When a TCP connection is closed for a socket using the AF_INET or AF_INET6 address family, the port associated with that connection is not made available until twice the Maximum Segment Life (MSL) time in seconds has passed. The MSL time is approximately 2 minutes. The SO_REUSEADDR option allows a *bind()* to succeed when requesting a port that is being held during this time frame. This can be especially useful if a server is abruptly ended and restarted.

   Notes:

   - For AF_INET and AF_INET6, SOCK_STREAM sockets, this option does **not** allow two servers to successfully issue a *bind()* requesting the same port number and local address combination. For AF_INET and AF_INET6, SOCK_DGRAM sockets, the SO_REUSEADDR option does allow multiple servers to successfully bind to the same port. When broadcast or multicast datagrams are

received for a given port, each server that is bound to that port receives a copy of the datagram provided each server has enabled the SO_REUSEADDR option.
   - This option does not affect unicast datagram delivery.
6. The following SOL_SOCKET options are not supported by AF_INET sockets over SNA. *setsockopt()* appears to succeed, but has no effect on the function of AF_INET sockets over SNA.
   - SO_BROADCAST
   - SO_DONTROUTE
   - SO_KEEPALIVE
   - SO_LINGER
7. The option IP_DONTFRAG » and IPV6_DONTFRAG are « not valid for multicast group destinations.
8. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *setsockopt()* API is mapped to *qso_setsockopt98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getsockopt()—Retrieve Information about Socket Options" on page 52—Retrieve Information about Socket Options

API introduced: V3R1

---

shutdown()—End Receiving and/or Sending of Data on Socket

BSD 4.3 Syntax

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

int shutdown(int socket_descriptor,
             int how)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int shutdown(int socket_descriptor,
             int how)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *shutdown()* function is used to disable reading, writing, or reading and writing on a socket.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

socket_descriptor
(Input) The descriptor of the socket to be shut down.

how (Input) The data flow path to be disabled:

`SHUT_RD or 0`	No more data can be received.
`SHUT_WR or 1`	No more data can be sent.
`SHUT_RDWR or 2`	No more data can be sent or received.

Authorities

No authorization is required.

Return Value

shutdown() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *shutdown()* fails, *errno* can be set to one of the following:

[EBADF]	Descriptor not valid.
[EINVAL]	Parameter not valid. This error code indicates one of the following: • The socket pointed to by the *socket_descriptor* parameter is using a connection-oriented transport service. Also, the transport service is in a state in which sends and receives are disallowed (for example, connection has been reset by peer). • The *how* parameter specifies a value that is not valid.
[ENOTCONN]	The specified descriptor does not reference a connected socket.
[ENOTSOCK]	The specified descriptor does not reference a socket.
[EIO]	Input/output error.
[EUNATCH]	The protocol required to support the specified address family is not available at this time. **Note:** This *errno* is not returned if the *how* parameter is 0.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. Issuing a *shutdown()* with a *how* parameter of 0 causes any new data received for the socket to be discarded. Any input functions for this socket complete with a 0, meaning that end-of-file has been

reached. » If the socket is being shared across multiple processes, any blocking input operations are deblocked by this action.

«

2. Issuing a *shutdown()* with a *how* parameter of 1 results in all output functions being failed with an error of [epipe]. The process issuing the output operation will receive a synchronous sigpipe signal. This also sends a normal close sequence to the partner program. Receive operations issued by the partner program receive a return value of 0 once all previous data has been received. » If the socket is being shared across multiple processes or threads, any blocking output functions are deblocked with a return value of -1 and an error code of [epipe].

«

3. Issuing a *shutdown()* with a *how* parameter of 2 results in the actions listed for a *how* parameter of 0 being performed first, followed by the actions listed for a *how* parameter of 1.
4. Issuing a *shutdown()* on socket connected through a *SOCKS server* is not supported.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *shutdown()* API is mapped to *qso_shutdown98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "close()—Close File or Socket Descriptor" on page 19—Close File or Socket Descriptor

API introduced: V3R1

---

socket()—Create Socket

Syntax

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

int socket(int address_family,
           int type,
           int protocol)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *socket()* function is used to create an end point for communications. The end point is represented by the socket descriptor returned by the *socket()* function.

Parameters

address_family

(Input) The address family to be used with the socket. Supported values are:

AF_INET	For interprocess communications between processes on the same system or different systems in the Internet domain using the Internet Protocol (IPv4).
AF_INET6	For interprocess communications between processes on the same system or different systems in the Internet domain using the Internet Protocol (IPv6 or IPv4).

AF_NS	For interprocess communications between processes on the same system or different systems in the domain defined by the Novell or Xerox protocol definitions. **Note:** The AF_NS address family is no longer supported as of V5R2.
AF_UNIX	For interprocess communications between processes on the same system in the UNIX domain.
AF_UNIX_CCSID	For interprocess communications between processes on the same system in the UNIX domain using the Qlg_Path_Name_T structure.
AF_TELEPHONY	For interprocess communications between processes on the same system in the telephony domain. **Note:** The AF_TELEPHONY address family is no longer supported as of V5R3.

type (Input) The type of communications desired. Supported values are:

SOCK_DGRAM	Indicates a datagram socket is desired.
SOCK_SEQPACKET	Indicates a full-duplex sequenced packet socket is desired. Each input and output operation consists of exactly one record.
SOCK_STREAM	Indicates a full-duplex stream socket is desired.
SOCK_RAW	Indicates communication is directly to the network protocols. A process must have the appropriate privilege *ALLOBJ to use this type of socket. Used by users who want to access the lower-level protocols directly.

protocol
(Input) The protocol to be used on the socket. Supported values are:

0	Indicates that the default protocol for the *type* selected is to be used. For example, IPPROTO_TCP is chosen for the protocol if the *type* was set to SOCK_STREAM and the address family is AF_INET.
IPPROTO_IP	Equivalent to specifying the value zero (0).
IPPROTO_TCP	Indicates that the TCP protocol is to be used.
IPPROTO_UDP	Indicates that the UDP protocol is to be used.
IPPROTO_RAW	Indicates that communications is to the IP layer.
IPPROTO_ICMP	Indicates that the Internet Control Message Protocol (ICMP) is to be used.
IPPROTO_ICMPV6	Indicates that the Internet Control Message Protocol (ICMPv6) is to be used.
TELPROTO_TEL	Equivalent to specifying the value zero (0).

Note: When the *type* is SOCK_RAW, the *protocol* can be set to some predefined protocol number from 0-255. See "Usage Notes" on page 180 for further details.

Authorities

When the SOCKET being created is of type SOCK_RAW, the thread must have *ALLOBJ special authority. When the thread does not have this authority, the EACCES is returned for *errno*.

Return Value

socket() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is a socket descriptor.

Error Conditions

When *socket()* fails, *errno* can be set to one of the following:

[EACCES]	Permission denied. Process does not have the appropriate privileges to create the socket with the specified type or protocol.
[EAFNOSUPPORT]	The type of socket is not supported in this protocol family.
[EIO]	Input/output error.
[EMFILE]	Too many descriptions for this process.
[ENFILE]	Too many descriptions in system.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[EPROTOTYPE]	The socket type or protocols are not compatible.
[EPROTONOSUPPORT]	No protocol of the specified type and domain exists.
[ESOCKTNOSUPPORT]	The specified socket type is not supported.
[EUNATCH]	The protocol required to support the specified address family is not available at this time.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The socket address families and types supported by sockets are defined in **<sys/socket.h>**. The protocols are defined in **<netinet/in.h>** (Internet protocols).
2. The `AF_UNIX` and `AF_UNIX_CCSID` address family supports a protocol of 0 for both `SOCK_STREAM` and `SOCK_DGRAM`.
3. The `AF_NS` address family is no longer supported as of V5R2.
4. The following tables list the combinations of types and protocols that are supported for `AF_INET` and the combinations of types and protocols that are supported for `AF_INET6`.

Supported Combinations of Types and Protocols for `AF_INET`	
Socket Type	**Protocol**
STREAM	`IPPROTO_TCP` (see Usage note 5)
DGRAM	`IPPROTO_UDP`
RAW	`IPPROTO_RAW`, `IPPROTO_ICMP`, *protocol_number*, (see Usage note 6)

Supported Combinations of Types and Protocols for `AF_INET6`	
Socket Type	**Protocol**
STREAM	`IPPROTO_TCP`
DGRAM	`IPPROTO_UDP`
RAW	`IPPROTO_RAW`, `IPPROTO_ICMPV6`, *protocol_number*, (see Usage note 6)

5. The **ALWANYNET** (Allow ANYNET support) network attribute allows a customer to select whether a SNA transport can be used for AF_INET socket applications.

The system administrator can see the current status of the **ALWANYNET** attribute and can change that status. (This can be done by using the Display Network Attributes (DSPNETA) and Change Network Attributes (CHGNETA) commands, respectively.)

If the status is changed, the change takes effect immediately. Also, the state of the **ALWANYNET** stays the same across IPLs. For example, if the current status is *YES and the system administrator changes the value to *NO, the use of AF_INET over a transport other than TCP/IP is deactivated. If a system IPL is performed after this point, the use of AF_INET over a SNA transport remains deactivated after the system IPL.

If AF_INET sockets will only be used over a TCP/IP transport, the **ALWANYNET** status should be set to *NO to improve CPU utilization.

Note: If you are also using APPC over TCP/IP **ALWANYNET** status needs to be set to *YES.

6. When the socket type is SOCK_RAW, you can specify any protocol number between 0-255. Two exceptions are the IPPROTO_TCP and IPPROTO_UDP protocols, which cannot be specified on a socket type of SOCK_RAW (if you issue *socket()*, you get an error with an error code of [EPROTONOSUPPORT]). Each raw socket is associated with one IP protocol number, and receives all data for that protocol. For example, if two processes create a raw socket with the same protocol number, and data is received for the protocol, then both processes get copies of the data.

   Protocol numbers 0 (IPPROTO_IP) and 255 (IPPROTO_RAW) have some unique characteristics. If a protocol number of zero is specified, then IP sends all data received from all the protocol numbers (except IPPROTO_TCP and IPPROTO_UDP protocols). If a protocol number of 255 is specified, a user must ensure that the IP header data is included in the data sent out on an output operation.
7. The AF_TELEPHONY address family is no longer supported as of V5R3.

Related Information

- "socketpair()—Create a Pair of Sockets"—Create a Pair of Sockets

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

socketpair()—Create a Pair of Sockets

BSD 4.3 Syntax

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

int socketpair(int address_family,
               int type,
               int protocol,
               int *socket_vector)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <sys/socket.h>

int socketpair(int address_family,
               int type,
               int protocol,
               int socket_vector[2])
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *socketpair()* function is used to create a pair of unnamed, connected sockets in the AF_UNIX or AF_UNIX_CCSID *address_family*.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

address_family
(Input) The address family to be used with the sockets. Supported values are:

AF_UNIX or AF_UNIX_CCSID	For interprocess communications between processes on the same system in the UNIX domain.

type (Input) The type of communications desired. Supported values are:

SOCK_DGRAM	Indicates a datagram socket is desired.
SOCK_STREAM	Indicates a full-duplex stream socket is desired.

protocol
(Input) The protocol to be used on the sockets. Supported values are:

0	Indicates the default protocol for the *type* selected is to be used.

socket_vector
(Output) An integer array of size two that will contain the socket descriptors.

Authorities

No authorization is required.

Return Value

socketpair() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When *socketpair()* fails, *errno* can be set to one of the following:

[EAFNOSUPPORT]	The type of socket is not supported in this protocol family.
[EFAULT]	Bad address.
[EINVAL]	Parameter not valid.
[EIO]	Input/output error.
[EMFILE]	Too many descriptions for this process.
[ENFILE]	Too many descriptions in system.
[ENOBUFS]	There is not enough buffer space for the requested operation.
[EOPNOTSUPP]	Operation not supported.
[EPROTONOSUPPORT]	No protocol of the specified type and domain exists.

[ESOCKTNOSUPPORT]	The specified socket type is not supported.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. The socket address families and types supported by sockets are defined in **<sys/socket.h>**.
2. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *socketpair()* API is mapped to *qso_socketpair98()*.
3. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), » it will fail with error code [ENOTSUP]. « See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "socket()—Create Socket" on page 178—Create Socket

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

takedescriptor()—Receive Socket Access from Another Job

Syntax

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

int takedescriptor(char *source_job)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *takedescriptor()* function is used to obtain a descriptor in one i5/OS job which was passed from another i5/OS job by a *givedescriptor()*.

Parameters

source_job

(Input) A pointer to the internal job identifier that identifies the source job from which to receive a passed descriptor.

Authorities

No authorization is required.

Return Value

takedescriptor() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is a descriptor.

Error Conditions

When *takedescriptor()* fails, *errno* can be set to one of the following:

[EFAULT]	Bad address. The system detected an address which was not valid while attempting to access the *source_job* parameter.
[EINVAL]	Parameter not valid. The *source_job* parameter points to data that is not valid.
[EMFILE]	Too many descriptions for this process.
[EIO]	Input/output error.
[EUNKNOWN]	Unknown system state.

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.

Usage Notes

1. This function can only obtain a descriptor if the sender of the descriptor referenced the job that this *takedescriptor()* is issued in by explicitly specifying this job's identification on the *target_job* parameter of the *givedescriptor()*.
2. If the *source_job* parameter is a NULL pointer, then a descriptor can be received from any job which issues a *givedescriptor()* that references the job in which *takedescriptor()* is issued.
3. If no descriptor is available to be received, the *takedescriptor()* is blocked.
4. If both the job in which the *givedescriptor()* is issued and the job specified by the *target_job* parameter end while a descriptor is in transit, the descriptor is reclaimed by the system, and the resource that it represents is closed.
5. The information to specify in the *target_job* parameter of the *givedescriptor()* and in the *source_job* parameter of the *takedescriptor()* can be obtained in the actual target job by using a work management API (for example, **QUSRJOBI**) to retrieve the *internal job identifier*.
6. For files and directories, *takedescriptor()* is only supported for objects in the Root, QOpenSys, User-defined file systems (UDFS), and Network File System (NFS).
7. If this function is called by a thread executing one of the scan-related exit programs (or any of its created threads), it will fail with error code [ENOTSUP]. See Integrated File System Scan on Open Exit Programs and Integrated File System Scan on Close Exit Programs for more information.
8. When the descriptor is obtained using *takedescriptor()*, any information accessed using that descriptor with the various read and write interfaces will be in binary, even if the original descriptor's accesses would have had text conversions occur. See Using CCSIDs and code pages in the open—Open file documentation for more information on text conversion.

Related Information

- "givedescriptor()—Pass Descriptor Access to Another Job" on page 60—Pass Descriptor Access to Another Job
- "sendmsg()—Send a Message Over a Socket" on page 146—Send Data or Descriptors or Both
- "recvmsg()—Receive a Message Over a Socket" on page 126—Receive Data or Descriptors or Both

API introduced: V3R1

---

write()—Write to Descriptor

Syntax

```
#include <unistd.h>

ssize_t write
  (int file_descriptor, const void *buf, size_t nbyte);
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 189.

The **write()** function writes *nbyte* bytes from *buf* to the file or socket associated with *file_descriptor*. *nbyte* should not be greater than INT_MAX (defined in the <**limits.h**> header file). If *nbyte* is zero, **write()** simply returns a value of zero without attempting any other action.

If *file_descriptor* refers to a "regular file" (a stream file that can support positioning the file offset) or any other type of file on which the job can do an **lseek()** operation, **write()** begins writing at the file offset associated with *file_descriptor*, unless O_APPEND is set for the file (see below). A successful **write()** increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

If O_APPEND (defined in the <**fcntl.h**> header file) is set for the file, **write()** sets the file offset to the end of the file before writing the output.

If there is not enough room to write the requested number of bytes (for example, because there is not enough room on the disk), the **write()** function writes as many bytes as the remaining space can hold.

If **write()** is successful and *nbyte* is greater than zero, the change and modification times for the file are updated.

If *file_descriptor* refers to a descriptor obtained using the **open()** function with O_TEXTDATA specified, the data is written to the file assuming it is in textual form. The maximum number of bytes on a single write that can be supported for text data is 2,147,483,408 (2GB - 240) bytes. The data is converted from the code page of the application, job, or system to the code page of the file as follows:

- When writing to a true stream file, any line-formatting characters (such as carriage return, tab, and end-of-file) are just converted from one code page to another.
- When writing to a record file that is being used as a stream file:
  - End-of-line characters are removed.
  - Records are padded with blanks (for a source physical file member) or nulls (for a data physical file member).
  - Tab characters are replaced by the appropriate number of blanks to the next tab position.

There are some important considerations if O_CCSID was specified on the **open()**.

- The **write()** will attempt to convert all of the data in the user's buffer. Successfully converted data will be written. Unconverted data is usually assumed to be a partial character. Partial characters will be buffered internally and data from the next consecutive write will be appended to the buffered data. If incorrect data is provided on a consecutive write, the write may fail with the [ECONVERT] error.

  If an **lseek()** is performed, the file is closed, or the current job is ended, the buffered data will be discarded. Discarded data will not be written to the file. See lseek()—Set File Read/Write Offset for more information.
- Because of the above consideration and because of the possible expansion or contraction of converted data, applications using the O_CCSID flag should avoid assumptions about data size and the current file offset. For example, the user may supply a buffer to 100 bytes, but after an application has written the buffer to a new file, the file size may be 50, 200, or something else, depending on the CCSIDs involved.

If O_TEXTDATA was not specified on the **open()**, the data is written to the file without conversion. The application is responsible for handling the data.

When *file_descriptor* refers to a socket, the **write()** function writes to the socket identified by the socket descriptor.

Note: When the write completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the write is unsuccessful, the bits are undefined.

Write requests to a pipe or FIFO are handled the same as a regular file, with the following exceptions:

- The S_ISUID and S_ISGID file mode bits will not be cleared.
- There is no file offset associated with a pipe or FIFO. Each write request will append to the end of the pipe or FIFO.
- Write requests of [PIPE_BUF] bytes or less will not be interleaved with data from other threads performing writes on the same pipe or FIFO. Writes of greater than [PIPE_BUF] bytes may have data interleaved on arbitrary boundaries with writes by other threads, whether or not the O_NONBLOCK flag of the file status flags is set.
- If the O_NONBLOCK flag was not specified and the pipe or FIFO is full, the write request will block the calling thread until the requested amount of data in *nbyte* is written.
- If the O_NONBLOCK flag was specified, then the following pertain to various write requests:

  – The **write()** function will not block the calling thread.
  – A write request for [PIPE_BUF] or fewer bytes will have the following effect:

    If there is sufficient space available in the pipe or FIFO, **write()** will transfer all the data and return the number of bytes requested. If there is not sufficient space in the pipe or FIFO, **write()** will transfer no data, return -1, and set *errno* to [EAGAIN].
  – A write request for more than [PIPE_BUF] bytes will cause one of the following:
    - When at least one byte can be written, **write()** will transfer what it can and return the number of bytes written.
    - When no data can be written, **write()** will transfer no data, return -1, and set *errno* to [EAGAIN].

Parameters

file_descriptor
(Input) The descriptor of the file to which the data is to be written.

buf (Input) A pointer to a buffer containing the data to be written.

nbyte (Input) The size in bytes of the data to be written.

Authorities

No authorization is required.

Return Value

value **write()** was successful. The value returned is the number of bytes actually written. This number is less than or equal to *nbyte*.

-1 **write()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **write()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. If writing to a socket, this error code indicates one of the following: • The destination address specified is a broadcast address and the socket option `SO_BROADCAST` was not set (with a *setsockopt()*). • The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with an address family of `AF_INET` and a type of `SOCK_DGRAM`.
[EAGAIN]	If *file_descriptor* refers to a pipe or FIFO that has its O_NONBLOCK flag set, this error occurs if the **write()** would have blocked the calling thread.
[EBADF] *[EBADFID]* *[EBUSY]* *[EDAMAGE]* *[EFAULT]*	
[EFBIG]	The size of the object would exceed the system allowed maximum size or the process soft file size limit. The file is a regular file, *nbyte* is greater than 0, and the starting offset is greater than or equal to 2 GB minus 2 bytes.
[EINTR]	
[EINVAL]	For example, the file system that the file resides in does not support large files, and the starting offset exceeds 2GB minus 2 bytes.
[EIO] *[EJRNDAMAGE]* *[EJRNENTTOOLONG]* *[EJRNINACTIVE]* *[EJRNRCVSPC]* *[ENEWJRN]* *[ENEWJRNRCV]* *[ENOMEM]* *[ENOSPC]* *[ENOTAVAIL]* *[ENOTSAFE]* *[ENXIO]* *[ERESTART]* *[ETRUNC]*	

Error condition	Additional information
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[EUNKNOWN]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNREFUSED]	This error code can only be returned on sockets that use a connectionless transport service.
[EDESTADDRREQ]	A destination address has not been associated with the socket pointed to by the *fildes* parameter. This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTDOWN]	This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTUNREACH]	This error code can only be returned on sockets that use a connectionless transport service.
[EINTR]	
[EMSGSIZE]	The data to be sent could not be sent atomically because the size specified by *nbyte* is too large.
[ENETDOWN]	This error code can only be returned on sockets that use a connectionless transport service.
[ENETUNREACH]	This error code can only be returned on sockets that use a connectionless transport service.
[ENOBUFS]	
[ENOTCONN]	This error code is returned only on sockets that use a connection-oriented transport service.
[EPIPE]	
[EUNATCH]	
[EWOULDBLOCK]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT]	
[EUNATCH]	

Error Messages

The following messages may be sent from this function:

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.

Message ID	Error Message Text
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:

   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

     – ″Root″ (/)
     – QOpenSys
     – User-defined
     – QNTC
     – QSYS.LIB
     – Independent ASP QSYS.LIB
     – QOPT
     – Network File System
     – QFileSvr.400

2. QSYS.LIB and independent ASP QSYS.LIB File System Differences

   This function will fail with error code [ENOTSAFE] if the object on which this function is operating is a save file and multiple threads exist in the job.

   If the file specified is a save file, only complete records will be written into the save file. A **write()** request that does not provide enough data to completely fill a save file record will cause the partial record's data to be saved by the file system. The saved partial record will then be combined with additional data on subsequent **write()**'s until a complete record may be written into the save file. If the save file is closed prior to a saved partial record being written into the save file, then the saved partial record is discarded, and the data in that partial record will need to be written again by the application.

   A successful **write()** updates the change, modification, and access times for a database member using the normal rules that apply to database files. At most, the access time is updated once per day.

   You should be careful when writing end-of-file characters in the QSYS.LIB and independent ASP QSYS.LIB file systems. These file systems end-of-file characters are symbolic; that is, they are stored outside the file member. However, some situations can result in actual, nonsymbolic end-of-file characters being written to a member. These nonsymbolic end-of-file characters could cause some tools or utilities to fail. For example:

   - If you previously wrote an end-of-file character as the last character of a member, do not continue to write data after that end-of-file character. Continuing to write data will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
   - If you previously wrote an end-of-file character as the last character of a member, do not write other end-of-file characters preceding it in the file. This will cause a nonsymbolic end-of-file to be written. As a result, a compile of the member could fail.
   - If you previously used the integrated file system interface to manipulate a member that contains an end-of-file character, avoid using other interfaces (such as the Source Entry Utility or database reads

and writes) to manipulate the member. If you use other interfaces after using the integrated file system interface, the end-of-file information will be lost.

3. QOPT File System Differences

   The change and modification times of the file are updated when the file is closed.

   When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.

4. Network File System Differences

   Local access to remote files through the Network File System may produce unexpected results due to conditions at the server. Once a file is open, subsequent requests to perform operations on the file can fail because file attributes are checked at the server on each request. If permissions on the file are made more restrictive at the server or the file is unlinked or made unavailable by the server for another client, your operation on an open file descriptor will fail when the local Network File System receives these updates. The local Network File System also impacts operations that retrieve file attributes. Recent changes at the server may not be available at your client yet, and old values may be returned from operations (several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data).

   Reading and writing to files with the Network File System relies on byte-range locking to guarantee data integrity. To prevent data inconsistency, use the **fcntl()** API to get and release these locks.

5. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.

6. Sockets Usage Notes

   a. **write()** only works with sockets on which a **connect()** has been issued, since it does not allow the caller to specify a destination address.
   b. To broadcast on an `AF_INET` socket, the socket option `SO_BROADCAST` must be set (with a **setsockopt()**).
   c. When using a connection-oriented transport service, all errors except `[EUNATCH]` and `[EUNKNOWN]` are mapped to `[EPIPE]` on an output operation when either of the following occurs:

      - A connection that is in progress is unsuccessful.
      - An established connection is broken.

      To get the actual error, use **getsockopt()** with the `SO_ERROR` option, or perform an input operation (for example, **read()**).

7. For the file systems that do not support large files, **write()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **write()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
8. Using this function successfully on the /dev/null or /dev/zero character special file results in a return value of the total number of bytes requested to be written. No data is written to the character special file. In addition, the change and modification times for the file are updated.
9. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <**fcntl.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- creat()—Create or Rewrite File
- dup()—Duplicate Open File Descriptor
- dup2()—Duplicate Open File Descriptor to Another Descriptor

- fclear()—Write (Binary Zeros) to Descriptor
- fclear64()—Write (Binary Zeros) to Descriptor (Large File Enabled)
- "fcntl()—Perform File Control Command" on page 28—Perform File Control Command
- "ioctl()—Perform I/O Control Request" on page 68—Perform I/O Control Request
- lseek()—Set File Read/Write Offset
- open()—Open File
- pread()—Read from Descriptor with Offset
- pread64()—Read from Descriptor with Offset (large file enabled)
- pwrite()—Write to Descriptor with Offset
- pwrite64()—Write to Descriptor with Offset (large file enabled)
- "read()—Read from Descriptor" on page 108—Read from Descriptor
- "readv()—Read from Descriptor Using Multiple Buffers" on page 114—Read from Descriptor Using Multiple Buffers
- "send()—Send Data" on page 143—Send Data
- "sendmsg()—Send a Message Over a Socket" on page 146—Send Data or Descriptors or Both
- "sendto()—Send Data" on page 153—Send Data
- "writev()—Write to Descriptor Using Multiple Buffers" on page 192—Write to Descriptor Using Multiple Buffers

Example

See Code disclaimer information for information pertaining to code examples.

The following example writes a specific number of bytes to a file:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define mega_string_len 1000000

main() {
  char *mega_string;
  int  file_descriptor;
  int  ret;
  char fn[]="write.file";

  if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
    perror("malloc() error");
  else if ((file_descriptor = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, '0', mega_string_len);
    if ((ret = write(file_descriptor, mega_string, mega_string_len)) == -1)
      perror("write() error");
    else printf("write() wrote %d bytes\n", ret);
    if (close(file_descriptor)!= 0)
       perror("close() error");
    if (unlink(fn)!= 0)
       perror("unlink() error");
  }
  free(mega_string);
}
```

Output:

```
write() wrote 1000000 bytes
```

API introduced: V3R1

writev()—Write to Descriptor Using Multiple Buffers

Syntax

```
#include <sys/types.h>
#include <sys/uio.h>

int writev(int descriptor,
           struct iovec *io_vector[],
           int vector_length)
```

Service Program Name: QP0LLIB1
Default Public Authority: *USE
Threadsafe: Conditional; see "Usage Notes" on page 194.

The **writev()** function is used to write data to a file or socket descriptor. **writev()** provides a way for the data that is going to be written to be stored in several different buffers (*scatter/gather I/O*).

Note: When the write completes successfully, the S_ISUID (set-user-ID) and S_ISGID (set-group-ID) bits of the file mode will be cleared. If the write is unsuccessful, the bits are undefined.

See "write()—Write to Descriptor" on page 185 for more information related to writing to a descriptor.

Parameters

descriptor
(Input) The descriptor to which the data is to be written. The descriptor refers to either a file or a socket.

io_vector[]
(Input) The pointer to an array of type **struct iovec**. **struct iovec** contains a sequence of pointers to buffers in which the data to be written is stored. The structure pointed to by the *io_vector* parameter is defined in **<sys/uio.h>**.

```
struct iovec {
   void      *iov_base;
   size_t   iov_len;
}
```

iov_base and *iov_len* are the only fields in *iovec* used by sockets. *iov_base* contains the pointer to a buffer and *iov_len* contains the buffer length. The rest of the fields are reserved.

vector_length
(Input) The number of entries in *io_vector*.

Authorities

No authorization is required.

Return Value

value	**writev()** was successful. The value returned is the number of bytes actually written.
-1	**writev()** was not successful. The *errno* global variable is set to indicate the error.

Error Conditions

If **writev()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

Error condition	Additional information
[EACCES]	If you are accessing a remote file through the Network File System, update operations to file permissions at the server are not reflected at the client until updates to data that is stored locally by the Network File System take place. (Several options on the Add Mounted File System (ADDMFS) command determine the time between refresh operations of local data.) Access to a remote file may also fail due to different mappings of user IDs (UID) or group IDs (GID) on the local and remote systems. If writing to a socket, this error code indicates one of the following: • The destination address specified is a broadcast address and the socket option `SO_BROADCAST` was not set (with a *setsockopt()*). • The process does not have the appropriate privileges to the destination address. This error code can only be returned on a socket with an address family of `AF_INET` and a type of `SOCK_DGRAM`.
[EAGAIN]	
[EBADF]	
[EBADFID]	
[EBUSY]	
[EDAMAGE]	
[EFAULT]	
[EFBIG]	The size of the object would exceed the system allowed maximum size or the process soft file size limit. The file is a regular file, *nbyte* is greater than 0, and the starting offset is greater than or equal to 2 GB minus 2 bytes.
[EINTR]	
[EINVAL]	For example, the file resides in a file system that does not support large files, and the starting offset exceeds 2GB minus 2 bytes.
[EIO]	
[EJRNDAMAGE]	
[EJRNENTTOOLONG]	
[EJRNINACTIVE]	
[EJRNRCVSPC]	
[ENEWJRN]	
[ENEWJRNRCV]	
[ENOMEM]	
[ENOSPC]	
[ENOTAVAIL]	
[ENOTSAFE]	
[ERESTART]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETRUNC]	
[EUNKNOWN]	

When the descriptor refers to a socket, *errno* could indicate one of the following errors:

Error condition	Additional information
[ECONNREFUSED]	This error code can only be returned on sockets that use a connectionless transport service.
[EDESTADDRREQ]	A destination address has not been associated with the socket pointed to by the *fildes* parameter. This error code can only be returned on sockets that use a connectionless transport service.

Error condition	Additional information
[EHOSTDOWN]	This error code can only be returned on sockets that use a connectionless transport service.
[EHOSTUNREACH]	This error code can only be returned on sockets that use a connectionless transport service.
[EINTR]	
[EMSGSIZE]	The data to be sent could not be sent atomically because the size specified by *nbyte* is too large.
[ENETDOWN]	This error code can only be returned on sockets that use a connectionless transport service.
[ENETUNREACH]	This error code can only be returned on sockets that use a connectionless transport service.
[ENOBUFS]	
[ENOTCONN]	This error code is returned only on sockets that use a connection-oriented transport service.
[EPIPE]	
[EUNATCH]	
[EWOULDBLOCK]	

If interaction with a file server is required to access the object, *errno* could indicate one of the following errors:

Error condition	Additional information
[EADDRNOTAVAIL]	
[ECONNABORTED]	
[ECONNREFUSED]	
[ECONNRESET]	
[EHOSTDOWN]	
[EHOSTUNREACH]	
[ENETDOWN]	
[ENETRESET]	
[ENETUNREACH]	
[ESTALE]	If you are accessing a remote file through the Network File System, the file may have been deleted at the server.
[ETIMEDOUT]	
[EUNATCH]	

Error Messages

Message ID	Error Message Text
CPE3418 E	Possible APAR condition or hardware failure.
CPF3CF2 E	Error(s) occurred during running of &1 API.
CPF9872 E	Program or service program &1 in library &2 ended. Reason code &3.
CPFA081 E	Unable to set return value or error code.
CPFA0D4 E	File system error occurred. Error number &1.

Usage Notes

1. This function will fail with error code [ENOTSAFE] when all the following conditions are true:
   - Where multiple threads exist in the job.
   - The object on which this function is operating resides in a file system that is not threadsafe. Only the following file systems are threadsafe for this function:

- ″Root″ (/)
- QOpenSys
- User-defined
- QNTC
- QSYS.LIB
- Independent ASP QSYS.LIB
- QOPT
- Network File System
- QFileSvr.400

2. **writev()** only works with sockets on which a *connect()* has been issued, since the call does not allow the caller to specify a destination address.
3. **writev()** is an atomic operation on sockets of type SOCK_DGRAM and SOCK_RAW in that it produces one packet of data every time it is issued. For example, a **writev()** to a datagram socket results in a single datagram.
4. To broadcast on an AF_INET socket, the socket option SO_BROADCAST must be set (with a *setsockopt()*).
5. When using a connection-oriented transport service, all errors except [EUNATCH] and [EUNKNOWN] are mapped to [EPIPE] on an output operation when either of the following occurs:
   - A connection that is in progress is unsuccessful.
   - An established connection is broken.

   To get the actual error, use *getsockopt()* with the SO_ERROR option, or perform an input operation (for example, *read()*).
6. For the file systems that do not support large files, **writev()** will return [EINVAL] if the starting offset exceeds 2GB minus 2 bytes, regardless of how the file was opened. For the file systems that do support large files, **writev()** will return [EFBIG] if the starting offset exceeds 2GB minus 2 bytes and the file was not opened for large file access.
7. QFileSvr.400 File System Differences

   The largest buffer size allowed is 16 megabytes. If a larger buffer is passed, the error EINVAL will be received.
8. QOPT File System Differences

   When writing to files on volumes formatted in Universal Disk Format (UDF), byte locks on the range being written are ignored.
9. Using this function successfully on the dev/null or /dev/zero character special file results in a return value of the total number of bytes requested to be written. No data is written to the character special file. In addition, the change and modification times for the file are updated.
10. If the write exceeds the process soft file size limit, signal SIFXFSZ is issued.

Related Information

- The <**fcntl.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- creat()—Create or Rewrite File
- dup()—Duplicate Open File Descriptor
- dup2()—Duplicate Open File Descriptor to Another Descriptor
- fclear()—Write (Binary Zeros) to Descriptor
- fclear64()—Write (Binary Zeros) to Descriptor (Large File Enabled)
- “fcntl()—Perform File Control Command” on page 28—Perform File Control Command
- “ioctl()—Perform I/O Control Request” on page 68—Perform I/O Control Request
- lseek()—Set File Read/Write Offset

- open()—Open File
- “read()—Read from Descriptor” on page 108—Read from Descriptor
- “readv()—Read from Descriptor Using Multiple Buffers” on page 114—Read from Descriptor Using Multiple Buffers
- “send()—Send Data” on page 143—Send Data
- “sendmsg()—Send a Message Over a Socket” on page 146—Send Data or Descriptors or Both
- “sendto()—Send Data” on page 153—Send Data
- “write()—Write to Descriptor” on page 185—Write to Descriptor

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

Sockets Network Functions

The network functions and the Berkeley Resolver routines supported by the sockets APIs are:

- “_getlong()—Get Long Byte Quantities” on page 334 (Get long byte quantities from a byte stream) is used to retrieve an unsigned long byte quantity.
- “_getshort()—Get Short Byte Quantities” on page 334 (Get short byte quantities from a byte stream.) is used to retrieve an unsigned short byte quantity.
- “_putlong()—Put Long Byte Quantities” on page 335 (Put long byte quantities into a byte stream.) is used to put an unsigned long byte quantity into a byte stream.
- “_putshort()—Put Short Byte Quantities” on page 336 (Put short byte quantities into a byte stream.) is used to put an unsigned short byte quantity into a byte stream.
- “dn_comp()—Compress Domain Name” on page 200 (Compress an expanded domain name) is used to compress an expanded domain name.
- “dn_comp_ts64()—Compress Domain Name” on page 202 (Compress an expanded domain name) is used to compress an expanded domain name.
- “dn_expand()—Expand Domain Name” on page 202 (Expand a compressed domain name.) is used to expand a compressed domain name.
- “dn_find()—Search for Compressed Domain Name” on page 204 (Search for a compressed domain name from a list of previously compressed domain names) is used to search for an expanded domain name in a list of compressed domain names.
- “dn_find_ts64()—Search for Compressed Domain Name” on page 205 (Search for a compressed domain name from a list of previously compressed domain names) is used to search for an expanded domain name in a list of compressed domain names.
- “dn_skipname()—Skip over Compressed Domain Name” on page 205 (Skip over a compressed domain name.) is used to skip over a compressed domain name in a DNS packet.
- “endhostent()—Close Host Database” on page 206 (Close the nameserver database) is used to close the host database file.
- “endhostent_r()—Close Host Database” on page 207 (Close the nameserver database) is used to close the host database file.
- “endnetent()—Close Network Database” on page 209 (Close the network database) is used to close the network database file.
- “endnetent_r()—Close Network Database” on page 210 (Close the network database) is used to close the network database file.
- “endprotoent()—Close Protocol Database” on page 211 (Close the protocol database) is used to close the protocols database file.
- “endprotoent_r()—Close Protocol Database” on page 212 (Close the protocol database) is used to close the protocol database file.

- "endservent()—Close Service Database" on page 213 (Close the service database) is used to close the services database file.
- "endservent_r()—Close Service Database" on page 214 (Close the service database) is used to close the service database file.
- "freeaddrinfo()—Free Address Information" on page 215 (Free Address Information) frees one or more addrinfo structures returned by getaddrinfo(), along with any additional storage associated with those structures.
- "gai_strerror()—Retrieve Address Information Runtime Error Message" on page 216 (Retrieve Address Information Runtime Error Message) retrieves a text string that describes a return value received from calling the getaddrinfo() or getnameinfo() API.
- "getaddrinfo()—Get Address Information" on page 217 (Get Address Information) translates the name of a service location or a service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.
- "gethostbyaddr()—Get Host Information for IP Address" on page 222 (Provide information about host given an Internet address) is used to retrieve information about a host.
- "gethostbyaddr_r()—Get Host Information for IP Address" on page 224 (Provide information about host given an Internet address) is used to retrieve information about a host.
- "gethostbyname()—Get Host Information for Host Name" on page 227 (Provide information about host given a host name) is used to retrieve information about a host.
- "gethostbyname_r()—Get Host Information for Host Name" on page 230 (Provide information about host given a host name) is used to retrieve information about a host.
- "gethostent()—Get Next Entry from Host Database" on page 233 (Get next host entry from the nameserver database) is used to retrieve information from the host database file.
- "gethostent_r()—Get Next Entry from Host Database" on page 235 (Get next host entry from the nameserver database) is used to retrieve information from the host database file.
- "getnameinfo()—Get Name Information for Socket Address" on page 236 (Get Name Information for Socket Address) translates a socket address to a node name and service location.
- "getnetbyaddr()—Get Network Information for IP Address" on page 239 (Get information from the network database about a given internet address) is used to retrieve information about a network.
- "getnetbyaddr_r()—Get Network Information for IP Address" on page 240 (Get information from the network database about a given internet address) is used to retrieve information about a network.
- "getnetbyname()—Get Network Information for Domain Name" on page 242 (Get information from the network database about a given domain name) is used to retrieve information about a network.
- "getnetbyname_r()—Get Network Information for Domain Name" on page 244 (Get information from the network database about a given domain name) is used to retrieve information about a network.
- "getnetent()—Get Next Entry from Network Database" on page 245 (Get network entry from the network database) is used to retrieve network information from the network database file.
- "getnetent_r()—Get Next Entry from Network Database" on page 246 (Get network entry from the network database) is used to retrieve network information from the network database file.
- "getprotobyname()—Get Protocol Information for Protocol Name" on page 248 (Get information regarding a protocol given the protocol name) is used to retrieve information about a protocol.
- "getprotobyname_r()—Get Protocol Information for Protocol Name" on page 250 (Get information regarding a protocol given the protocol name) is used to retrieve information about a protocol.
- "getprotobynumber()—Get Protocol Information for Protocol Number" on page 251 (Get information regarding a protocol given the protocol number) is used to retrieve information about a protocol.
- "getprotobynumber_r()—Get Protocol Information for Protocol Number" on page 253 (Get information regarding a protocol given the protocol number) is used to retrieve information about a protocol.
- "getprotoent()—Get Next Entry from Protocol Database" on page 254 (Get next protocol entry in the protocol data base) is used to retrieve protocol information from the protocol database file.

- “getprotoent_r()—Get Next Entry from Protocol Database” on page 255 (Get next protocol entry in the protocol data base) is used to retrieve protocol information from the protocol database file.
- “getservbyname()—Get Port Number for Service Name” on page 257 (Get port number for a given service name.) is used to retrieve information about services (the protocol being used by the service and the port number assigned for the service).
- “getservbyname_r()—Get Port Number for Service Name” on page 259 (Get port number for a given service name.) is used to retrieve information about services: the protocol being used by the service and the port number assigned for the service.
- “getservbyport()—Get Service Name for Port Number” on page 261 (Get service name given a port number) is used to retrieve information about a service assigned to a port number.
- “getservbyport_r()—Get Service Name for Port Number” on page 262 (Get service name given a port number) is used to retrieve information about a service assigned to a port number.
- “getservent()—Get Next Entry from Service Database” on page 264 (Get next service entry from the service database) is used to retrieve information about services (the protocol being used by the service and the port number assigned for the service).
- “getservent_r()—Get Next Entry from Service Database” on page 265 (Get next service entry from the service database) is used to retrieve information about services: the protocol being used by the service and the port number assigned for the service.
- “hstrerror()—Retrieve Resolver Error Message” on page 267 (Retrieve resolver error message.) is used to retrieve the text string that describes a resolver h_errno value.
- “htonl()—Convert Long Integer to Network Byte Order” on page 268 (Convert a long (4 byte) integer from local host byte order to the network byte order) is used to convert a long (4-byte) integer from the local host byte order to standard network byte order.
- “htons()—Convert Short Integer to Network Byte Order” on page 269 (Convert a short (2 byte) integer from local host byte order to the network byte order) is used to convert a short (2-byte) integer from the local host byte order to standard network byte order.
- “inet_addr()—Translate Full Address to 32-bit IP Address” on page 270 (Translate the full address from dotted decimal format to a 32-bit Internet address) is used to translate an Internet address from dotted decimal format to a 32-bit IP address.
- “inet_lnaof()—Separate Local Portion of IP Address” on page 272 (Separate the local portion of an Internet address.) is used to extract the local host portion of an IP address.
- “inet_makeaddr()—Combine Network Portion and Host Portion to Make IP Address” on page 273 (Formulate an Internet address that combines a network address with the local address of a host.) is used to generate a 32-bit IP address from the 32-bit network IP address and the local address of the host.
- “inet_netof()—Separate Network Portion of IP Address” on page 274 (Separate the network portion of an Internet address.) is used to extract the network portion of an IP address.
- “inet_network()—Translate Network Portion of Address to 32-bit IP Address” on page 275 (Translate the network portion of the address from dotted decimal format to a 32-bit Internet address) is used to translate an Internet address from dotted decimal format to a 32-bit network IP address, in which the host part of the IP address is set to zeros.
- “inet_ntoa()—Translate IP Address to Dotted Decimal Format” on page 277 (Translate from 32-bit Internet address to a dotted decimal format) is used to translate an Internet address from a 32-bit IP address to dotted decimal format.
- “inet_ntoa_r()—Translate IP Address to Dotted Decimal Format” on page 277 (Translate from 32-bit Internet address to a dotted decimal format) is used to translate an Internet address from a 32-bit IP address to dotted decimal format.
- “inet_ntop()—Convert IPv4 and IPv6 Addresses Between Binary and Text Form” on page 278 (Convert IPv4 and IPv6 Addresses Between Binary and Text Form) converts a numeric address into a text string suitable for presentation.

- "inet_pton()—Convert IPv4 and IPv6 Addresses Between Text and Binary Form" on page 280 (Convert IPv4 and IPv6 Addresses Between Text and Binary Form) converts an address in its standard text presentation form into its numeric binary form.
- "ns_addr()—Translate Network Services Address to 12-byte Address" on page 282 (Translate a network services address from human readable format to a 12-byte hexadecimal address) is used to translate a network services address from human readable format to a 12-byte hexadecimal address.
- "ns_ntoa()—Translate Network Services Address from 12-byte Address/h2>" on page 283 (Translate a network services address from a 12-byte address to a human readable format) is used to translate a network services address from a 12-byte address to a human readable format.
- "ns_ntoa_r() — Translate Network Services Address from 12-byte Address" on page 284 (Translate a network services address from a 12-byte address to a human readable format) is used to translate a network services address from a 12-byte address to a human readable format.
- "ntohl()—Convert Long Integer to Host Byte Order" on page 285 (Convert a long (4 byte) integer from network byte order to the local host byte order) is used to convert a long (4-byte) integer from the standard network byte order to the local host byte order.
- "ntohs()—Convert Short Integer to Host Byte Order" on page 286 (Convert a short (2 byte) integer from network byte order to the local host byte order) is used to convert a short (2-byte) integer from the standard network byte order to the local host byte order.
- "res_close()—Close Socket and Reset _res Structure" on page 287 (Close a socket and reset the _res structure.) is used to reset the _res structure to the beginning defaults and close a socket that is opened as a result of the RES_STAYOPEN flag.
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288 (Find the enclosing zone and servers) queries name servers until it finds the enclosing zone and its master name servers for the specified domain name.
- "res_hostalias()—Retrieve the host alias" on page 291 (Retrieve the host alias) looks up the specified name in the host aliases file specified by the environment variable HOSTALIASES.
- "res_init()—Initialize _res Structure" on page 292 (Initialize _res structure for domain name server.) is used to initialize the _res structure for name resolution.
- "res_mkquery()—Place Domain Query in Buffer" on page 296 (Form a domain name query and place it in a buffer in memory.) is used to make standard query messages (DNS packets) for name servers.
- "res_nclose()—Close Socket and Reset res Structure" on page 299 (Close socket and reset res structure) is used to reset the _res structure to the beginning defaults and close a socket that is opened as a result of the RES_STAYOPEN flag.
- "res_ninit()—Initialize res Structure" on page 299 (Initialize res structure) is used to initialize the _res structure for name resolution.
- "res_nisourserver()—Check Server Address" on page 304 (Check server address) looks up the specified server address in the ns_addr_list[] of the specified res structure.
- "res_nmkquery()—Place Domain Query in Buffer" on page 305 (Place domain query in buffer) is used to make standard query messages (DNS packets) for name servers.
- "res_nmkupdate()—Construct an Update Packet" on page 306 (Construct an update packet) builds a dynamic update packet from the linked list of update records.
- "res_nquery()—Send Domain Query" on page 307 (Send domain query) is used to interface to the server query mechanism.
- "res_nquerydomain()—Send 2 String Domain Query" on page 308 (Send 2-string domain query) is used to interface to the server query mechanism.
- "res_nsearch()—Search for Domain Name" on page 309 (Search for domain name) is used to make a query message and wait for a response.
- "res_nsend()—Send Buffered Domain Query or Update" on page 310 (Send buffered domain query or update) is used to send a query or update message to a name server and retrieve a response.

- "res_nsendsigned()—Send Authenticated Domain Query or Update" on page 311 (Send authenticated domain query or update) is similar to res_nsend() but it uses the specified key to create a transaction signature (TSIG) to sign the query or update packet and to authenticate the response.
- "res_nupdate()—Build and Send Dynamic Updates" on page 314 (Build and send dynamic updates) separates the linked list of update records into groups so that all records in a group will belong to a single zone on the nameserver.
- "res_query()—Send Domain Query" on page 316 (Form a domain name query and send it to the domain name server.) is used to interface to the server query mechanism.
- "res_search()—Search for Domain Name" on page 318 (Search for a domain name from a list of domain names) is used to make a query message and wait for a response.
- "res_send()—Send Buffered Domain Query or Update" on page 320 (Send the query formed in res_mkquery to the domain name server.) is used to send a query or update message to a name server and retrieve a response.
- "res_xlate()—Translate DNS Packets" on page 323 (Translate standard DNS packets between ASCII and EBCDIC) is used to translate a standard DNS packet between ASCII and EBCDIC.
- "sethostent()—Open Host Database" on page 325 (Open the nameserver database) is used to prepare for sequential access to the host database file. sethostent() opens the file and repositions the file marker to the beginning of the file.
- "sethostent_r()—Open Host Database" on page 326 (Open the nameserver database) is used in preparation for sequential access to the host database file.
- "setnetent()—Open Network Database" on page 327 (Open the network database) is used to prepare for sequential access to the network database file.
- "setnetent_r()—Open Network Database" on page 328 (Open the network database) is used in preparation for sequential access to the network database file.
- "setprotoent()—Open Protocol Database" on page 329 (Open the protocol database) is used to prepare for sequential access to the protocol database file.
- "setprotoent_r()—Open Protocol Database" on page 330 (Open the protocol database) is used in preparation for sequential access to the protocol database file.
- "setservent()—Open Service Database" on page 331 (Open the service database) is used to prepare for sequential access to the service database file.
- "setservent_r()—Open Service Database" on page 332 (Open the service database) is used in preparation for sequential access to the service database file.

[1] IBM(R) addition to the Berkeley Resolver Routines

Note: These functions use header (include) files from the library QSYSINC, which is optionally installable. Make sure QSYSINC is installed on your system before using any of the functions.

Top | UNIX-Type APIs | APIs by category

dn_comp()—Compress Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp(unsigned char *expanded_domain_name,
            unsigned char *compressed_domain_name,
            int  answer_buffer_length,
            unsigned char **domain_name_pointers,
            unsigned char **last_domain_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *dn_comp()* function is used to compress an expanded domain name.

Authorities and Locks

None.

Parameters

expanded_domain_name
(Input) The pointer to the expanded domain name.

compressed_domain_name
(Output) The pointer to where the compressed domain name will be stored.

answer_buffer_length
(Input) The size of the *compressed_domain_name* buffer.

domain_name_pointers
(Input) The pointer to an array of pointers to previously compressed domain names in the current message.

last_domain_name
(Input) The pointer to the end of the array specified by *domain_name_pointers*.

Return Value

dn_comp() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the size of the compressed domain name.

  dn_comp() compresses the domain name pointed to by *expanded_domain_name*. The result is placed in *compressed_domain_name*.

Error Conditions

When the *dn_comp()* function fails, it does not set specific *errno* or *h_errno* values. An error occurs under the following conditions:

- NULL pointer(s) passed to the function.
- Invalid pointer(s) passed to the function.
- *Compressed_domain_name* too small for the compressed domain name.

Usage Notes

1. *domain_name_pointers*[0] points to the beginning of the DNS packet. The list of pointers ends with a NULL pointer. After *domain_name_pointers*[0] is initialized to the beginning of the packet and *domain_name_pointers*[1] is initialized to NULL, *dn_comp()* updates the list each time it is called.
2. *dn_comp()* calls *dn_find()* to attempt to locate the different parts of the domain name being compressed.
3. *dn_comp()* expects EBCDIC data as input. The output from *dn_comp()* is also EBCDIC.

Related Information

- "dn_expand()—Expand Domain Name" on page 202—Expand Domain Name
- "dn_find()—Search for Compressed Domain Name" on page 204-Search for Compressed Domain Name

- "dn_skipname()—Skip over Compressed Domain Name" on page 205—Skip over Compressed Domain Name

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

dn_comp_ts64()—Compress Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp_ts64(unsigned char * __ptr64 expanded_domain_name,
            unsigned char * __ptr64 compressed_domain_name,
            int  answer_buffer_length,
            unsigned char * __ptr64 * __ptr64 domain_name_pointers,
            unsigned char * __ptr64 * __ptr64 last_domain_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *dn_comp_ts64()* function is used to compress an expanded domain name. *dn_comp_ts64()* differs from *dn_comp()* in that *dn_comp_ts64()* accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, and other related information, see "dn_comp()—Compress Domain Name" on page 200—Compress Domain Name.

Usage Notes

All of the usage notes for "dn_comp()—Compress Domain Name" on page 200—Compress Domain Name apply to *dn_comp_ts64()*.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

dn_expand()—Expand Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand(unsigned char *message_pointer,
              unsigned char *end_of_message,
              unsigned char *compressed_domain_name,
              unsigned char *expanded_domain_name,
              int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *dn_expand()* function is used to expand a compressed domain name.

Authorities and Locks

None.

Parameters

message_pointer
(Input) The pointer to the beginning of a DNS packet.

end_of_message
(Input) The pointer to the end of the DNS packet.

compressed_domain_name
(Input) The pointer to the compressed domain name within the DNS packet.

expanded_domain_name
(Output) The pointer to the expanded domain name.

answer_buffer_length
(Input) The size of the *expanded_domain_name* buffer.

Return Value

dn_expand() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the size of the compressed domain name.

  The *dn_expand()* routine expands the domain name pointed to by *compressed_domain_name*. The result is placed in *expanded_domain_name*.

Error Conditions

When the *dn_expand()* function fails, it does not set specific *errno* or *h_errno* values. An error occurs under the following conditions:

- NULL pointer(s) passed to the function.
- Invalid pointer(s) passed to the function.
- *expanded_domain_name* too small for the expanded domain name.
- *end_of_message* reached before the domain name could be expanded.

Usage Notes

1. The compressed domain name size is returned rather than the expanded domain name size because it is used to parse through the DNS packet.
2. *dn_expand()* uses *end_of_message* to insure that it doesn't run past the end of the DNS packet.
3. *dn_expand()* expects EBCDIC data as input. The output from *dn_expand()* is also EBCDIC.

Related Information

- "dn_comp()—Compress Domain Name" on page 200—Compress Domain Name
- "dn_find()—Search for Compressed Domain Name" on page 204—Search for Compressed Domain Name
- "dn_skipname()—Skip over Compressed Domain Name" on page 205—Skip over Compressed Domain Name

API introduced: V3R1

dn_find()—Search for Compressed Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_find(unsigned char *expanded_domain_name,
            unsigned char *message_pointer,
            unsigned char **domain_name_pointers,
            unsigned char **last_domain_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *dn_find()* function is used to search for an expanded domain name in a list of compressed domain names.

Authorities and Locks

None.

Parameters

expanded_domain_name
(Input) The pointer to the expanded domain name.

message_pointer_name
(Input) A pointer to the DNS packet that contains the compressed names pointed to by the elements of **domain_name_pointers**.

domain_name_pointers
(Input) The pointer to an array of pointers to previously compressed names in the current message.

last_domain_name
(Input) The pointer to the end of the array of *domain_name_pointers*.

Return Value

dn_find() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is an offset into the *message_pointer* where domain name was found.

Error Conditions

When the *dn_find()* function fails, it does not set specific *errno* or *h_errno* values. An error occurs under the following conditions:

- NULL pointer(s) passed to the function.
- Invalid pointer(s) passed to the function.
- Expanded domain name not found in the DNS packet.

Usage Notes

1. *dn_find()* locates an expanded name in an array of previously compressed names.
2. Usually *dn_find()* is called from *dn_comp()* but can be called directly.
3. *dn_find()* expects EBCDIC data as input.

Related Information

- "dn_expand()—Expand Domain Name" on page 202—Expand Domain Name
- "dn_comp()—Compress Domain Name" on page 200—Compress Domain Name
- "dn_skipname()—Skip over Compressed Domain Name"—Skip over Compressed Domain Name

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

dn_find_ts64()—Search for Compressed Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_find_ts64(unsigned char * __ptr64 expanded_domain_name,
            unsigned char * __ptr64 message_pointer,
            unsigned char * __ptr64 * __ptr64 domain_name_pointers,
            unsigned char * __ptr64 * __ptr64 last_domain_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *dn_find()* function is used to search for an expanded domain name in a list of compressed domain names. *dn_find_ts64()* differs from *dn_find()* in that *dn_find_ts64()* accepts 8-byte teraspace pointers.

For a discussion of the parameters, authorities required, return values, and other related information, see "dn_find()—Search for Compressed Domain Name" on page 204—Search for Compressed Domain Name.

Usage Notes

All of the usage notes for "dn_find()—Search for Compressed Domain Name" on page 204—Compress Domain Name apply to *dn_find_ts64()*.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

dn_skipname()—Skip over Compressed Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
```

```
#include <resolv.h>

int dn_skipname(unsigned char *compressed_domain_name,
                unsigned char *end_of_message)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *dn_skipname()* function is used to skip over a compressed domain name in a DNS packet.

Authorities and Locks

None.

Parameters

compressed_domain_name
(Input) A pointer to a compressed domain name.

end_of_message
(Input) The pointer to the end of the message string.

Return Value

dn_skipname() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the size of *compressed_domain_name*.

Error Conditions

When the *dn_skipname()* function fails, it does not set specific *errno* or *h_errno* values. An error occurs under the following conditions:

- NULL pointer(s) passed to the function.
- Invalid pointer(s) passed to the function.
- *end_of_message* reached before the end of the compressed domain name.

Usage Notes

1. *dn_skipname()* skips over a compressed domain name in a DNS packet and returns the size of *compressed_domain_name*.
2. *dn_skipname()* expects EBCDIC data as input.

Related Information

- “dn_expand()—Expand Domain Name” on page 202—Expand Domain Name
- “dn_find()—Search for Compressed Domain Name” on page 204—Search for Compressed Domain Name
- “dn_comp()—Compress Domain Name” on page 200—Compress Domain Name

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

endhostent()—Close Host Database

Syntax

```
#include <netdb.h>

void endhostent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *endhostent()* function is used to close the host database file. The file is opened by those functions that retrieve information about a host (for example, *gethostent()*).

Authorities

No authorization is required.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the host file is always closed. When the _XOPEN_SOURCE macro is not so defined, the host file is not closed if a *sethostent()* with a nonzero parameter value was previously completed.
2. iSeries Navigator or the following CL commands can be used to access the host database file:
   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
3. Do not use the *endhostent()* function in a multithreaded environment. See the multithread alternative *endhostent_r()* function.
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endhostent()* API is mapped to *qso_endhostent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "gethostent()—Get Next Entry from Host Database" on page 233—Get Next Entry from Host Database
- "gethostbyname()—Get Host Information for Host Name" on page 227—Get Host Information for Host Name
- "gethostbyaddr()—Get Host Information for IP Address" on page 222—Get Host Information for IP Address
- "sethostname()—Set Host Name" on page 165—Open Host Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

endhostent_r()—Close Host Database

Syntax

```
#include <netdb.h>

void endhostent_r(struct hostent_data
                  *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *endhostent_r()* function is used to close the host database file. The file is opened by those functions that retrieve information about a host (for example, *gethostent_r()*).

Parameters

struct hostent_data *hostent_data_struct_addr (input)

Specifies the pointer to the hostent_data structure, which is used to pass and preserve results between function calls. The field host_control_blk in the hostent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire hostent_data structure must be initialized to hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *endhostent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct hostent_data**denoted by ***hostent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *endhostent_r()* function fails, *errno* can be set to:

[EINVAL]

The hostent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure hostent_data.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the host file is always closed. When the _XOPEN_SOURCE macro is not so defined, the host file will not be closed if a *sethostent_r()* call with a nonzero parameter value was previously done.
2. The iSeries Navigator or the following CL commands can be used to access the host database file:

   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endhostent_r()* API is mapped to *qso_endhostent_r98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "gethostbyaddr_r()—Get Host Information for IP Address" on page 224—Get Host Information for IP Address

- "gethostbyname_r()—Get Host Information for Host Name" on page 230—Get Host Information for Host Name
- "gethostent_r()—Get Next Entry from Host Database" on page 235—Get Next Entry from Host Database
- "sethostent_r()—Open Host Database" on page 326—Open Host Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

endnetent()—Close Network Database

Syntax

```
#include <netdb.h>

void endnetent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *endnetent()* function is used to close the network database file. The file is opened by those functions that retrieve information about a network (for example, *getnetent()*).

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the network file is always closed. When the _XOPEN_SOURCE macro is not so defined, the network file is not closed if a *setnetent()* with a nonzero parameter value was previously completed.
2. The iSeries Navigator or the following CL commands can be used to access the network database file:
   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
3. Do not use the *endnetent()* function in a multithreaded environment. See the multithread alternative *endnetent_r()* function.
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endnetent()* API is mapped to *qso_endnetent98()*.

Authorities

No authorization is required.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getnetent()—Get Next Entry from Network Database" on page 245—Get Next Entry from Network Database
- "setnetent()—Open Network Database" on page 327—Open Network Database
- "getnetbyaddr()—Get Network Information for IP Address" on page 239-Get Network Information for IP Address
- "getnetbyname()—Get Network Information for Domain Name" on page 242—Get Network Information for Domain Name

API introduced: V3R1

endnetent_r()—Close Network Database

Syntax

```
#include <netdb.h>
int endnetent_r(struct netent_data
                *netent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *endnetent_r()* function is used to close the network database file. The file is opened by those functions that retrieve information about a network (for example, *getnetent_r()*).

Parameters

struct netent_data *netent_data_struct_addr (input)

Specifies the pointer to the netent_data structure, which is used to pass and preserve results between function calls. The field net_control_blk in the netent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire netent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *endnetent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct netent_data**denoted by ***netent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *endnetent_r()* function fails, *errno* can be set to:

[EINVAL]

The netent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure netent_data.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the network file is always closed. When the _XOPEN_SOURCE macro is not so defined, the network file will not be closed if a *setnetent_r()* call with a nonzero parameter value was previously done.
2. The iSeries Navigator or the following CL commands can be used to access the network database file:

   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)

3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endnetent_r()* API is mapped to*qso_endnetent_r98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getnetent_r()—Get Next Entry from Network Database" on page 246—Get Next Entry from Network Database
- "getnetbyaddr_r()—Get Network Information for IP Address" on page 240—Get Network Information for IP Address
- "getnetbyname_r()—Get Network Information for Domain Name" on page 244—Get Network Information for Domain Name
- "setnetent_r()—Open Network Database" on page 328—Open Network Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

endprotoent()—Close Protocol Database

Syntax

```
#include <netdb.h>

void endprotoent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *endprotoent()* function is used to close the protocols database file. The file is opened by those functions that retrieve information about a protocol (for example, *getprotoent()*).

Authorities

No authorization is required.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the protocols file is always closed. When the _XOPEN_SOURCE macro is not so defined, the protocols file is not closed if a *setprotoent()* with a nonzero parameter value was previously completed.
2. The iSeries Navigator or the following CL commands can be used to access the protocol database file:
   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
3. Do not use the *endprotoent()* function in a multithreaded environment. See the multithread alternative *endprotoent_r()* function.
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endprotoent()* API is mapped to *qso_endprotoent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

- “getprotoent()—Get Next Entry from Protocol Database” on page 254—Get Next Entry from Protocol Database
- “setprotoent()—Open Protocol Database” on page 329—Open Protocol Database
- “getprotobyname()—Get Protocol Information for Protocol Name” on page 248—Get Protocol Information for Protocol Name
- “getprotobynumber()—Get Protocol Information for Protocol Number” on page 251—Get Protocol Information for Protocol Number

API introduced: V3R1

---

endprotoent_r()—Close Protocol Database

Syntax

```
#include <netdb.h>
int endprotoent_r(struct protoent_data
                  *protoent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *endprotoent_r()* function is used to close the protocol database file. The file is opened by those functions that retrieve information about a protocol (for example, *getprotoent_r()*).

Parameters

struct protoent_data *protoent_data_struct_addr (input)

Specifies the pointer to the protoent_data structure, which is used to pass and preserve results between function calls. The field proto_control_blk must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire protoent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *endprotoent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct protoent_data** denoted by ***protoent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *endprotoent_r()* function fails, *errno* can be set to:

[EINVAL]

The protoent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure protoent_data.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the protocols file is always closed. When the _XOPEN_SOURCE macro is not so defined, the protocols file will not be closed if a *setprotoent_r()* call with a non-zero parameter value was previously done.
2. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endprotoent_r()* API is mapped to *qso_endprotoent_r98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getprotobynumber_r()—Get Protocol Information for Protocol Number" on page 253—Get Protocol
- "getprotobyname_r()—Get Protocol Information for Protocol Name" on page 250—Get Protocol Information for Protocol Name
- "getprotoent_r()—Get Next Entry from Protocol Database" on page 255—Get Next Entry from Protocol Database
- "setprotoent_r()—Open Protocol Database" on page 330—Open Protocol Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

endservent()—Close Service Database

Syntax

```
#include <netdb.h>

void endservent()
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *endservent()* function is used to close the services database file. The file is opened by those functions that retrieve information about services (for example, *getservent()*).

Authorities

No authorization is required.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the services file is always closed. When the _XOPEN_SOURCE macro is not so defined, the services file is not closed if a *setservent()* with a nonzero parameter value was previously completed.
2. The iSeries Navigator or the following CL commands can be used to access the services database file:
   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)

3. Do not use the *endservent()* function in a multithreaded environment. See the multithread alternative *endservent_r()* function.
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endservent()* API is mapped to *qso_endservent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getservent()—Get Next Entry from Service Database" on page 264—Get Next Entry from Service Database
- "setservent()—Open Service Database" on page 331—Open Service Database
- "getservbyname()—Get Port Number for Service Name" on page 257—Get Port Number for Service Name
- "getservbyport()—Get Service Name for Port Number" on page 261—Get Service Name for Port Number

API introduced: V3R1

---

endservent_r()—Close Service Database

Syntax

```
#include <netdb.h>
int endservent_r(struct servent_data
                 *servent_data_struct_addr)
```

Service Program Name: QSOSRV1
Default Public Authority: *USE
Threadsafe: Yes

The *endservent_r()* function is used to close the service database file. The file is opened by those functions that retrieve information about services (for example, *getservent_r()*).

Parameters

struct servent_data *_servent_data_struct_addr_ (input)
Specifies the pointer to the servent_data structure, which is used to pass and preserve results between function calls. The field serve_control_blk in the servent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire servent_data structure must initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *endservent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct servent_data**denoted by ***servent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *endservent_r()* function fails, *errno* can be set to:

[EINVAL]

The servent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action,see the description for structure servent_data.

Usage Notes

1. When the _XOPEN_SOURCE macro defined to the value 520 or greater, the services file is always closed. When the _XOPEN_SOURCE macro is not so defined, the services file will not be closed if a *setservent_r()* call with a non-zero parameter value was previously done.
2. The iSeries Navigator or the following CL commands can be used to access the services database file:

   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *endservent_r()* API is mapped to *qso_endservent_r98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getservbyname_r()—Get Port Number for Service Name” on page 259—Get Port Number for Service Name
- “getservbyport_r()—Get Service Name for Port Number” on page 262—Get Service Name for Port Number
- “getservent_r()—Get Next Entry from Service Database” on page 265—Get Next Entry from Service Database
- “setservent_r()—Open Service Database” on page 332—Open Service Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

freeaddrinfo()—Free Address Information

Syntax

```
#include  <sys/socket.h>
#include  <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The **freeaddrinfo()** function frees one or more addrinfo structures returned by “getaddrinfo()—Get Address Information” on page 217, along with any additional storage associated with those structures. If the *ai_next* field of the structure is not null, the entire list of structures is freed.

Parameters

ai (Input) The pointer to a **struct addrinfo** that was returned by "getaddrinfo()—Get Address Information" on page 217.

The structure **struct addrinfo** is defined in **<netdb.h>**.

```
struct addrinfo {
  int     ai_flags;     /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, .. */
  int     ai_family;    /* PF_xxx */
  int     ai_socktype;  /* SOCK_xxx */
  int     ai_protocol;  /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
  socklen_t ai_addrlen; /* length of ai_addr */
  char   *ai_canonname; /* canonical name for nodename */
  struct sockaddr  *ai_addr; /* binary address */
  struct addrinfo  *ai_next; /* next structure in linked list */
};
```

Authorities

No authorization is required.

Usage Notes

1. The **freeaddrinfo()** API supports the freeing of arbitrary sublists of an *addrinfo* list originally returned by "getaddrinfo()—Get Address Information" on page 217.

Related Information

- "getaddrinfo()—Get Address Information" on page 217—Get Address Information

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

gai_strerror()—Retrieve Address Information Runtime Error Message

Syntax

```
#include  <sys/socket.h>
#include  <netdb.h>

char *gai_strerror(int ecode);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***gai_strerror()*** function retrieves a text string that describes a return value received from calling the "getaddrinfo()—Get Address Information" on page 217 or "getnameinfo()—Get Name Information for Socket Address" on page 236 API.

Parameters

ecode (Input) The return value received from "getaddrinfo()—Get Address Information" on page 217 or "getnameinfo()—Get Name Information for Socket Address" on page 236.

Authorities

No authorization is required.

Return Value

gai_strerror() returns a pointer to the return value text.

Usage Notes

1. **gai_strerror()** returns a pointer to the string. The null-terminated string is stored in the CCSID of the job. If the job is 65535 and the string is something other than EBCDIC single byte or EBCDIC mixed, the text is converted to the default job CCSID.
2. If an *ecode* is specified for which there is no corresponding description, an Unknown Error string is returned.
3. The null-terminated string addressed by the pointer returned is overlayed by subsequent invocations of the **gai_strerror()** API from within the same thread.

Related Information

- "getaddrinfo()—Get Address Information"—Get Address Information
- "getnameinfo()—Get Name Information for Socket Address" on page 236—Get Name Information for Socket Address

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

getaddrinfo()—Get Address Information

Syntax

```
#include  <sys/socket.h>
#include  <netdb.h>

int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The **getaddrinfo()** function translates the name of a service location (for example, a host name) and/or a service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

Parameters

The *nodename* and *servname* parameters are either null pointers or pointers to null-terminated strings. One or both of these two parameters must be a non-null pointer.

The format of a valid name depends on the protocol family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the implementation will attempt to resolve the name in all supported families and, in the absence of errors, one or more results shall be returned.

nodename

(Input) The pointer to the null-terminated character string that contains the descriptive name or address string for which the address information is to be retrieved. If the **servname** parameter is null, a **nodename** must be specified and the requested network-level address will be returned. If

the **nodename** parameter is null, a **servname** must be specified and the requested service location will be assumed to be local to the caller. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the permissable address string formats for the *nodename* parameter are specified as defined in "inet_pton()—Convert IPv4 and IPv6 Addresses Between Text and Binary Form" on page 280

servname

(Input) The pointer to the null-terminated character string that contains the descriptive name or numeric representation suitable for use with the address family or families for which the requested service information is to be retrieved. If nodename is not null, the requested service location is named by **nodename**; otherwise, the requested service location is local to the caller. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

hints

(Input) The pointer to a **struct addrinfo**. If the parameter **hints** is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this hints structure every member other than *ai_flags*, *ai_family*, *ai_socktype* and *ai_protocol* must be zero or a null pointer. If hints is a null pointer, the behavior will be as if it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype* and *ai_protocol* fields, and AF_UNSPEC for the *ai_family* field.

The structure **struct addrinfo** is defined in **<netdb.h>**.

```
struct addrinfo {
  int     ai_flags;     /* AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, .. */
  int     ai_family;    /* PF_xxx */
  int     ai_socktype;  /* SOCK_xxx */
  int     ai_protocol;  /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
  socklen_t ai_addrlen; /* length of ai_addr */
  char   *ai_canonname; /* canonical name for nodename */
  struct sockaddr  *ai_addr; /* binary address */
  struct addrinfo  *ai_next; /* next structure in linked list */
};
```

A value of AF_UNSPEC for *ai_family* means that the caller will accept any protocol family. A value of zero for *ai_socktype* means that the caller will accept any socket type. A value of zero for *ai_protocol* means that the caller will accept any protocol.

If the caller handles only IPv4 and not IPv6, then the *ai_family* member of the hints structure should be set to PF_INET when **getaddrinfo()** is called.

If the caller handles only TCP and not UDP, for example, then the *ai_protocol* member of the hints structure should be set to IPPROTO_TCP when **getaddrinfo()** is called.

The *ai_flags* field to which hints parameter points must have the value zero or be the bitwise OR of one or more of the values AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, AI_NUMERICSERV, AI_V4MAPPED, AI_ALL, and AI_ADDRCONFIG.

The AI_PASSIVE flag in the *ai_flags* member of the hints structure specifies how to fill in the IP address portion of the socket address structure. If the AI_PASSIVE flag is specified, then the returned address information will be suitable for use in binding a socket for accepting incoming connections for the specified service (that is, a call to "bind()—Set Local Address for Socket" on page 13). In this case, if the **nodename** parameter is null, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set, the returned address information will be suitable for a call to "connect()—Establish Connection or Destination Address" on page 22 (for a connection-oriented protocol) or for a call to "connect()—Establish

Connection or Destination Address" on page 22, "sendto()—Send Data" on page 153 or "sendmsg()—Send a Message Over a Socket" on page 146 (for a connectionless protocol). In this case, if the **nodename** parameter is null, then the IP address portion of the socket address structure will be set to the loopback address. This flag is ignored if the **nodename** parameter is not null.

If the flag AI_CANONNAME is specified and the **nodename** parameter is not null, the function attempts to determine the canonical name corresponding to nodename (for example, if **nodename** is an alias or shorthand notation for a complete name).

If the flag AI_NUMERICHOST is specified then a non-null **nodename** string must be a numeric host address string. Otherwise an error of [EAI_NONAME] is returned. This flag prevents any type of name resolution service (for example, the DNS) from being called.

If the flag AI_NUMERICSERV is specified then a non-null **servname** string must be a numeric port string. Otherwise an error [EAI_NONAME] is returned. This flag prevents any type of name resolution service (for example, NIS+) from being called.

If the AI_V4MAPPED flag is specified along with an *ai_family* of AF_INET6, then the caller will accept IPv4-mapped IPv6 addresses. That is, if no AAAA records are found then a query is made for A records and any found are returned as IPv4-mapped IPv6 addresses (*ai_addrlen* will be 28). The AI_V4MAPPED flag is ignored unless *ai_family* equals AF_INET6.

The AI_ALL flag is used in conjunction with the AI_V4MAPPED flag, and is only used with an *ai_family* of AF_INET6. When AI_ALL is logically or'd with AI_V4MAPPED flag then the caller will accept all addresses: IPv6 and IPv4-mapped IPv6. A query is first made for AAAA records and if successful, the IPv6 addresses are returned. Another query is then made for A records and any found are returned as IPv4-mapped IPv6 addresses (*ai_addrlen* will be 28). This flag is ignored unless *ai_family* equals AF_INET6.

If the AI_ADDRCONFIG flag is specified then a query for AAAA records will occur only if the node has at least one IPv6 source address configured and a query for A records will occur only if the node has at least one IPv4 source address configured. The loopback address is not considered for this case as valid as a configured source address.

The *ai_socktype* field to which argument hints points specifies the socket type for the service. If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation will attempt to resolve the service name for all supported socket types and, all successful results will be returned. A non-zero socket type value will limit the returned information to values with the specified socket type.

res (Output) The pointer to a linked list of **addrinfo** structures, each of which specifies a socket address and information for use in creating a socket with which to use that socket address. The list will include at least one **addrinfo** structure. The *ai_next* field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list includes values for use with a call to the "socket()—Create Socket" on page 178 function, and a socket address for use with the "connect()—Establish Connection or Destination Address" on page 22 function or, if the AI_PASSIVE flag was specified, for use with the "bind()—Set Local Address for Socket" on page 13 function. The fields *ai_family*, *ai_socktype*, and *ai_protocol* are usable as the arguments to the "socket()—Create Socket" on page 178 function to create a socket suitable for use with the returned address. The fields *ai_addr* and *ai_addrlen* are usable as the arguments to the "connect()—Establish Connection or Destination Address" on page 22 or "bind()—Set Local Address for Socket" on page 13 functions with such a socket, according to the AI_PASSIVE flag.

If nodename is not null, and if requested by the AI_CANONNAME flag, the *ai_canonname* field of the first returned addrinfo structure points to a null-terminated string containing the canonical name corresponding to the input nodename; if the canonical name is not available, then *ai_canonname* refers to the argument nodename or a string with the same contents. The contents of the *ai_flags* field of the returned structures is undefined.

All fields in socket address structures returned by **getaddrinfo()** that are not filled in through an explicit argument (for example, *sin6_flowinfo* and *sin_zero*) will be set to zero.

Note: This makes it easier to compare socket address structures.

Authorities

Authorization of *R (allow access to the object) to the host aliases file specified by the *HOSTALIASES* environment variable.

You also need *X authority to each directory in the path of the host aliases file.

Return Value

getaddrinfo() returns an integer. Possible values are:

- 0 (successful)
- non-zero (unsuccessful)

Error Conditions

When *getaddrinfo()* fails, the error return value can be set to one of the following:

[EAI_AGAIN]

The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS]

The flags parameter had an invalid value.

[EAI_FAIL]

A non-recoverable error occurred when attempting to resolve the name.

[EAI_FAMILY]

The address family was not recognized.

[EAI_MEMORY]

There was a memory allocation failure when trying to allocate storage for the return value.

[EAI_NONAME]

The name does not resolve for the supplied parameters. Neither **nodename** nor **servname** were passed. At least one of these must be passed.

[EAI_SERVICE]

The service passed was not recognized for the specified socket type.

[EAI_SOCKTYPE]

The intended socket type was not recognized.

[EAI_SYSTEM]

A system error occurred; the error code can be found in *errno*

Usage Notes

1. The "freeaddrinfo()—Free Address Information" on page 215 API **must** be used to free the *addrinfo* structures returned by **getaddrinfo()**.
2. The "gai_strerror()—Retrieve Address Information Runtime Error Message" on page 216 API may be used to retrieve an error message associated with one of the error return values described above.
3. A job has a coded character set identifier (CCSID) and a default CCSID. The default CCSID is the same as the job CCSID unless the job CCSID specifies 65535, which requests that no database translation be performed. In this case, the default CCSID is set by the system based on the language ID in effect for the job.

   If the address information is retrieved from the domain name server, sockets converts the address information specified by the *nodename* and *servname* parameters from the default (CCSID) to ASCII before communicating with the domain name server. If the address information is retrieved from the host database file, no conversion is done on the node and service names specified by the *nodename* and *servname* parameters unless the CCSID of the job is something other than 65535.

   In addition, the canonical names for nodename returned in the addrinfo structures will be returned in the default CCSID of the job if they are obtained from the domain name server. For conversion to occur for the canonical names returned in the addrinfo structures when they are obtained from the host database file, you must use a job CCSID of something other than 65535.
4. The host database file currently only supports IPv4 addresses.
5. » *getaddrinfo()* has been extended to allow scope zone name or scope zone index to be appended to *nodename*. For example:

   ```
   www.ibm.com%8 or www.ibm.com%ethline
   FE80::1%8 or FE80::1%ethline
   ```

   If appended, the *sin6_scope_id* field in the *sockaddr_in6* structure pointed to by *ai_addr* will be set to the interger value associated with the appended value.

   «
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getaddrinfo()* API is mapped to *qetaddrinfo98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "bind()—Set Local Address for Socket" on page 13—Set a Local Address for the Socket
- "connect()—Establish Connection or Destination Address" on page 22—Establish Connection or Destination Address
- "freeaddrinfo()—Free Address Information" on page 215—Free Address Information
- "gai_strerror()—Retrieve Address Information Runtime Error Message" on page 216—Retrieve Address Information Runtime Error Message
- "gethostbyname()—Get Host Information for Host Name" on page 227—Get Host Information for Host Name
- "getnameinfo()—Get Name Information for Socket Address" on page 236—Get Name Information for Socket Address
- "getservbyname()—Get Port Number for Service Name" on page 257—Get Port Number for Service Name
- "getservbyport()—Get Service Name for Port Number" on page 261—Get Service Name for Port Number
- "inet_pton()—Convert IPv4 and IPv6 Addresses Between Text and Binary Form" on page 280—Convert IPv4 and IPv6 Addresses Between Text and Binary Form
- "sendto()—Send Data" on page 153—Send Data

- "sendmsg()—Send a Message Over a Socket" on page 146—Send Data or Descriptors or Both
- "socket()—Create Socket" on page 178—Create a Socket

API introduced: V5R2

---

gethostbyaddr()—Get Host Information for IP Address

BSD 4.3 Syntax

```
#include <netdb.h>

struct hostent *gethostbyaddr(char *host_address,
                              int address_length,
                              int address_type)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 223.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct hostent *gethostbyaddr(const void *host_address,
                              socklen_t address_length,
                              int address_type)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 223.

The *gethostbyaddr()* function is used to retrieve information about a host.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

host_address
(Input) The pointer to a structure of type **in_addr** that contains the address of the host for which information is to be retrieved.

address_length
(Input) The length of the *host_address*.

address_type
(Input) The domain type of the host address. `AF_INET` is the only value for this parameter that is supported.

Authorities

No authorization is required.

Return Value

gethostbyaddr() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct hostent**, defined in **<netdb.h>**.

```
struct hostent {
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
};

#define h_addr  h_addr_list[0]
```

h_name points to the character string that contains the name of the host. *h_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the host. *h_addrtype* contains the address type of the host (for example, AF_INET). *h_length* contains the address length. *h_addr_list* is a pointer to a NULL-terminated list of pointers, each of which points to a network address for the host, in network byte order. Note that the array of address pointers points to structures of type **in_addr** defined in **<netinet/in.h>**.

Error Conditions

When *gethostbyaddr()* fails, *h_errno* (defined in **<netdb.h>**) can be set to one of the following:

[HOST_NOT_FOUND]

The host name specified by the *host_address* parameter was not found.

[NO_DATA]

The host name is a valid name, but there is no corresponding IP address.

[NO_RECOVERY]

An unrecoverable error has occurred.

[TRY_AGAIN]

The local server did not receive a response from an authoritative server. An attempt at a later time may succeed.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:

   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. The pointer returned by *gethostbyaddr()* points to static storage that is overwritten on subsequent calls to the *gethostbyaddr()*, *gethostbyname()*, or *gethostent()* functions.
3. There are two sources from which host information can be obtained: the domain name server, and the host database file. The path taken depends on whether an IP address is configured for a name server using the iSeries Navigator or option 12, Change TCP/IP domain information, on the Configure TCP/IP (CFGTCP) menu.

Note: A person with a UNIX background would expect this information to exist in a file known as **/etc/resolv.conf**. If the IP address is found (indicating that the local network is a domain network), the *gethostbyaddr()* function attempts to query the domain name server for information about a host. If the query fails, the information is obtained from the host database file. If the name server IP address is not found (indicating that local network is a flat network), the host database file is used to obtain the host information.

4. When host information is retrieved from the host database file, the opened file is only closed if a *sethostent()* with a nonzero parameter value was not previously done.
5. If a *sethostent()* with a nonzero parameter value was previously done, *gethostbyaddr()*, when obtaining host information from the domain name server, communicates with the domain name server over a connection-oriented transport service (for example, TCP). Otherwise, *gethostbyaddr()* uses a connectionless transport service (for example, UDP).
6. If the host information is obtained from the domain name server, the information is returned in the default coded character set identifier (CCSID) currently in effect for the job. (The default CCSID is the same as the job CCSID unless 65535 is requested, in which case the default CCSID is set based on the language ID of the job. See globalization for more information.) If the host information is retrieved from the host database file, the default CCSID of the job is not used. To request translation of the host information when it is retrieved from the host database file, you must use a job CCSID of something other than 65535.
7. Address families are defined in **<sys/socket.h>**, and the **in_addr** structure is defined in **<netinet/in.h>**.
8. Do not use the *gethostbyaddr()* function in a multithreaded environment. See the multithread alternative *gethostbyaddr_r()* function.
9. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *gethostbyaddr()* API is mapped to *qso_gethostbyaddr98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “hstrerror()—Retrieve Resolver Error Message” on page 267—Retrieve Resolver Error Message
- “res_hostalias()—Retrieve the host alias” on page 291—Retrieve the host alias
- “gethostbyname()—Get Host Information for Host Name” on page 227—Get Host Information for Host Name
- “gethostent()—Get Next Entry from Host Database” on page 233—Get Next Entry from Host Database
- “sethostent()—Open Host Database” on page 325—Open Host Database
- “endhostent()—Close Host Database” on page 206—Close Host Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

gethostbyaddr_r()—Get Host Information for IP Address

BSD 4.3 Syntax

```
#include <netdb.h>

int gethostbyaddr_r(char *host_address,
                    int address_length,
                    int address_type,
                    struct hostent *hostent_struct_addr,
                    struct hostent_data *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

int gethostbyaddr_r(const void *host_address,
                    socklen_t address_length,
                    int address_type,
                    struct hostent *hostent_struct_addr,
                    struct hostent_data *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *gethostbyaddr_r()* function is used to retrieve information about a host.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

host_address (input)
Specifies the pointer to a structure of type **in_addr** that contains the address of the host for which information is to be retrieved.

address_length (input)
Specifies the length of the *host_address*.

address_type (input)
Specifies the domain type of the host address. Currently, `af_inet` is the only value for this parameter that is supported.

hostent_struct_addr (input/output)
Specifies the pointer to a hostent structure where the results will be placed. All results must be referenced through this structure.

hostent_data_struct_addr (input/output)
Specifies the pointer to the hostent_data structure, which is used to pass and preserve results between function calls. The field host_control_blk in the hostent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire hostent_data structure must initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *gethostbyaddr_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct hostent** denoted by ***hostent_struct_addr*** and **struct hostent_data**denoted by ***hostent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct hostent**is defined as:

```
struct hostent [
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
];

#define h_addr  h_addr_list[0]
```

h_name points to the character string that contains the name of the host. *h_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the host. *h_addrtype* contains the address type of the host (for example, `af_inet`). *h_length* contains the size of an address in octets (for example, the size of an Internet address is 4 octets). *h_addr_list* is a pointer to a NULL-terminated list of pointers, each of which points to a network address (in network byte order) for the host.

Error Conditions

When the *gethostbyaddr_r()* function fails, *h_errno* (defined in **<netdb.h>**) can be set to:

[HOST_NOT_FOUND]

The host name specified by the *host_address* parameter was not found.

[NO_DATA]

The host name is a valid name, but there is no corresponding IP address.

[NO_RECOVERY]

An unrecoverable error has occurred.

[TRY_AGAIN]

The local server did not receive a response from an authoritative server. An attempt at a later time may succeed.

When the *gethostbyaddr_r()* function fails, *errno* can be set to:

[EINVAL]

The hostent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure hostent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:
   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. There are two sources from which host information can be obtained: the domain name server and the host database file. The path taken depends on whether an IP address is configured for a name server using the iSeries Navigator or option 12, Change TCP/IP domain information, on the **CFGTCP** menu.

   Note: A person with a UNIX background would expect this information to exist in a file known as **/etc/resolv.conf**. If the IP address is found (indicating that the local network is a domain network), the *gethostbyaddr_r()* function will attempt to query the domain name server for information about a host. If the query fails, the information will be obtained from the host database file. If the name server IP address is not found (indicating that local network is a flat network), the host database file is used to obtain the host information.

3. When the host information is obtained from the host database file, the file is opened and the host information is retrieved (if it exists) from the file. The file is then closed only if a *sethostent_r()* call with a non-zero parameter value was not previously done.
4. If a *sethostent_r()* call with a non-zero parameter value was previously done, the *gethostbyaddr_r()* routine, when obtaining host information from the domain name server, will communicate with the domain name server over a connection-oriented transport service (for example, TCP). Otherwise, *gethostbyaddr_r()* will use a connectionless transport service (for example, UDP).
5. If the host information is obtained from the domain name server, the information is returned in the default coded character set identifier (CCSID) currently in effect for the job. (The default CCSID is the same as the job CCSID unless 65535 is requested, in which case the default CCSID is set based on the language ID of the job. See the globalization topic for more information.) If the host information is retrieved from the host database file the default CCSID of the job is not used. To request translation of the host information when it is retrieved from the host database file, you must use a job CCSID of something other than 65535.
6. Address families are defined in **<sys/socket.h>**, and the **in_addr** structure is defined in **<netinet/in.h>**.
7. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *gethostbyaddr_r()* API is mapped to *qso_gethostbyaddr_r98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "gethostbyname_r()—Get Host Information for Host Name" on page 230—Get Host Information for Host Name
- "gethostent_r()—Get Next Entry from Host Database" on page 235—Get Next Entry from Host Database
- "endhostent_r()—Close Host Database" on page 207—Close Host Database
- "sethostent_r()—Open Host Database" on page 326—Open Host Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

gethostbyname()—Get Host Information for Host Name

BSD 4.3 Syntax

```
#include <netdb.h>

struct hostent *gethostbyname(char *host_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 229.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct hostent *gethostbyname(const char *host_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see “Usage Notes” on page 229.

The *gethostbyname()* function is used to retrieve information about a host.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

host_name

(Input) The pointer to the character string that contains the name of the host for which information is to be retrieved.

Authorities

Authorization of *R (allow access to the object) to the host aliases file specified by the *HOSTALIASES* environment variable.

You also need *X authority to each directory in the path of the host aliases file.

Return Value

gethostbyname() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct hostent**.

The structure **struct hostent** is defined in **<netdb.h>**.

```
struct hostent {
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
};

#define h_addr  h_addr_list[0]
```

h_name points to the character string that contains the name of the host. *h_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the host. *h_addrtype* contains the address type of the host (for example, `AF_INET`). *h_length* contains the address length. *h_addr_list* is a pointer to a NULL-terminated list of pointers, each of which points to a network address for the host, in network byte order. Note that the array of address pointers points to structures of type **in_addr** defined in **<netinet/in.h>**.

Error Conditions

When *gethostbyname()* fails, *h_errno* (defined in **<netdb.h>**) can be set to one of the following:

[HOST_NOT_FOUND]

The host name specified by the *host_name* parameter was not found.

[NO_DATA]

The host name is a valid name, but there is no corresponding IP address.

[NO_RECOVERY]

An unrecoverable error has occurred.

[TRY_AGAIN]

The local server did not receive a response from an authoritative server. An attempt at a later time may succeed.

When the *gethostbyname()* function fails, *errno* can be set to:

[EACCES]

Permission denied. The process does not have the appropriate privileges to the host aliases file specified by the *HOSTALIASES* environment variable.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:

   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. The pointer returned by *gethostbyname()* points to static storage that is overwritten on subsequent calls to the *gethostbyname()*, *gethostbyaddr()*, or *gethostent()* functions.
3. There are two sources from which host information can be obtained: the domain name server, and the host database file. The path taken depends on whether an IP address is configured for a name server using the iSeries Navigator or option 12, Change TCP/IP domain information, on the Configure TCP/IP (CFGTCP) menu.

   Note: A person with a UNIX background would expect this information to exist in a file known as **/etc/resolv.conf**.

   If the IP address is found (indicating that the local network is a domain network), the *gethostbyaddr()* function attempts to query the domain name server for information about a host. If the query fails, the information is obtained from the host database file. If the name server IP address is not found (indicating that local network is a flat network), the host database file is used to obtain the address.
4. If the *host_name* parameter does specify a domain qualified name, the *gethostbyaddr()* function appends a domain name to the specified host name, if possible. The domain name that is appended is configured using the iSeries Navigator or CFGTCP menu option 12, Change TCP/IP domain information.
5. When the host information is obtained from the host database file, the file is opened and the host information is retrieved (if it exists) from the file. The file is then closed only if a *sethostent()* with a nonzero parameter value was not previously done.
6. If a *sethostent()* with a nonzero parameter value was previously done, the *gethostbyname()* routine, when obtaining host information from the domain name server, communicates with the domain name server over a connection-oriented transport service (for example, TCP). Otherwise, *gethostbyname()* uses a connectionless transport service (for example, UDP).
7. A job has a coded character set identifier (CCSID) and a default CCSID. The default CCSID is the same as the job CCSID unless the job CCSID specifies 65535, which requests that no database translation be performed. In this case, the default CCSID is set by the system based on the language ID in effect for the job.

   If the host information is retrieved from the domain name server, sockets converts the host name specified by the *host_name* parameter from the default (CCSID) to ASCII before communicating with the domain name server. If the host information is retrieved from the host database file, no

conversion is done on the host name specified by the *host_name* parameter unless the CCSID of the job is something other than 65535. In addition, the host names returned in the hostent structure will be returned in the default CCSID of the job if they are obtained from the domain name server. For translation to occur for the host names returned in the hostent structure when they are obtained from the host database file, you must use a job CCSID of something other than 65535.

8. Address families are defined in **<sys/socket.h>**, and the **in_addr** structure is defined in **<netinet/in.h>**.
9. Do not use the *gethostbyname()* function in a multithreaded environment. See the multithread alternative *gethostbyname_r()* function.
10. *gethostbyname()* will resolve local host aliases to a domain name which are then resolved with a query using DNS. See "res_hostalias()—Retrieve the host alias" on page 291 for more information on aliases.
11. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *gethostbyname()* API is mapped to *qso_gethostbyname98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "gethostbyaddr()—Get Host Information for IP Address" on page 222—Get Host Information for IP Address
- "gethostent()—Get Next Entry from Host Database" on page 233—Get Next Entry from Host Database
- "sethostent()—Open Host Database" on page 325—Open Host Database
- "endhostent()—Close Host Database" on page 206—Close Host Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

gethostbyname_r()—Get Host Information for Host Name

BSD 4.3 Syntax

```
#include <netdb.h>

int gethostbyname_r(char *host_name,
                    struct hostent *hostent_struct_addr,
                    struct hostent_data *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

int gethostbyname_r(const char *host_name,
                     struct hostent *hostent_struct_addr,
                     struct hostent_data *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *gethostbyname_r()* function is used to retrieve information about a host.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

host_name (input)
Specifies the pointer to the character string that contains the name of the host for which information is to be retrieved.

hostent_struct_addr (input/output)
Specifies the pointer to a hostent structure where the results will be placed. All results must be referenced through this structure.

hostent_data_struct_addr (input/output)
Specifies the pointer to the hostent_data structure, which is used to pass and preserve results between function calls. The field host_control_blk in the hostent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire hostent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities:

Authorization of *R (allow access to the object) to the host aliases file specified by the *hostaliases* environment variable.

You also need *X authority to each directory in the path of the host aliases file.

Return Value

The *gethostbyname_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct hostent** denoted by ***hostent_struct_addr*** and **struct hostent_data**denoted by ***hostent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct hostent**is defined as:

```
struct hostent [
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
];

#define h_addr  h_addr_list[0]
```

h_name points to the character string that contains the name of the host. *h_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the host. *h_addrtype* contains the address type of the host (for example, `af_inet`). *h_length* contains the size of an address in octets (for example, the size of an Internet address is 4 octets). *h_addr_list* is a pointer to a NULL-terminated list of pointers, each of which points to a network address (in network byte order) for the host.

Error Conditions

When the *gethostbyname_r()* function fails, *h_errno* (defined in **<netdb.h>**) can be set to:

[HOST_NOT_FOUND]

The host name specified by the *host_name* parameter was not found.

[NO_DATA]

The host name is a valid name, but there is no corresponding IP address.

[NO_RECOVERY]

An unrecoverable error has occurred.

[TRY_AGAIN]

The local server did not receive a response from an authoritative server. An attempt at a later time may succeed.

When the *gethostbyname_r()* function fails, *errno* can be set to:

[EACCES]

Permission denied. The process does not have the appropriate privileges to the host aliases file specified by the *HOSTALIASES* environment variable.

[EINVAL]

The hostent_data structure was not initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure hostent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:

   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. There are two sources from which host information can be obtained: the domain name server and the host database file. The path taken depends on whether an IP address is configured for a name server using the iSeries Navigator or option 12, Change TCP/IP domain information, on the **CFGTCP** menu.

   Note: A person with a UNIX background would expect this information to exist in a file known as **/etc/resolv.conf**. If the IP address is found (indicating that the local network is a domain network), the *gethostbyaddr_r()* function will attempt to query the domain name server for information about a host. If the query fails, the information will be obtained from the host database file. If the name server IP address is not found (indicating that local network is a flat network), the host database file is used to obtain the address.
3. If the *host_name* parameter does specify a domain qualified name, the *gethostbyaddr_r()* function will append a domain name to the specified host name, if possible. The domain name that will be appended is configured using the iSeries Navigator or **CFGTCP** menu option 12, Change TCP/IP domain information.
4. When the host information is obtained from the host database file, the file is opened and the host information is retrieved (if it exists) from the file. The file is then closed only if a *sethostent_r()* call with a non-zero parameter value was not previously done.
5. If a *sethostent_r()* call with a non-zero parameter value was previously done, the *gethostbyname_r()* routine, when obtaining host information from the domain name server, will communicate with the

domain name server over a connection-oriented transport service (for example, TCP). Otherwise, *gethostbyname_r()* will use a connectionless transport service (for example, UDP).

6. A job has a coded character set identifier (CCSID) and a default CCSID. The default CCSID is the same as the job CCSID unless the job CCSID specifies 65535, which requests that no database translation be performed. In this case, the default CCSID is set by the system based on the language ID in effect for the job.

   If the host information is retrieved from the domain name server, sockets converts the host name specified by the *host_name* parameter to ASCII before communicating with the domain name server. If the host information is retrieved from the host database file, no conversion is done on the host name specified by the *host_name* parameter unless the CCSID of the job is something other than 65535. In addition, host names returned in the hostent will be returned in the default CCSID of the job if they are obtained from the domain name server. For translation to occur for the host names returned in the hostent structure when they are obtained from the host database file, you must use a job CCSID of something other than 65535.
7. Address families are defined in **<sys/socket.h>**, and the **in_addr** structure is defined in **<netinet/in.h>**.
8. *gethostbyname_r()* will resolve local host aliases to a domain name which are then resolved with a query using DNS. See "res_hostalias()—Retrieve the host alias" on page 291 for more information on aliases.
9. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *gethostbyname_r()* API is mapped to *qso_gethostbyname_r98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "endhostent_r()—Close Host Database" on page 207—Close Host Database
- "gethostbyaddr_r()—Get Host Information for IP Address" on page 224—Get Host Information for IP Address
- "gethostent_r()—Get Next Entry from Host Database" on page 235—Get Next Entry from Host Database
- "sethostent_r()—Open Host Database" on page 326—Open Host Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

gethostent()—Get Next Entry from Host Database

Syntax

```
#include <netdb.h>

struct hostent *gethostent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 234.

The *gethostent()* function is used to retrieve information from the host database file. When *gethostent()* is first called, the file is opened, and the first entry is returned. Each subsequent call to *gethostent()* results in the next entry in the file being returned. To close the file, use *endhostent()*.

Authorities

No authorization is required.

Return Value

gethostent() returns a pointer. Possible values are:

- NULL (unsuccessful or end-of-file)
- p (successful), where p is a pointer to **struct hostent**.

The structure **struct hostent** is defined in **<netdb.h>**.

```
struct hostent {
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
};

#define h_addr  h_addr_list[0]
```

h_name points to the character string that contains the name of the host. *h_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the host. *h_addrtype* contains the address type of the host (for example, `AF_INET`). *h_length* contains the address length. *h_addr_list* is a pointer to a NULL-terminated list of pointers, each of which points to a network address for the host, in network byte order. Note that the array of address pointers points to structures of type **in_addr** defined in **<netinet/in.h>**.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:
   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. The pointer returned by *gethostent()* points to static storage that is overwritten on subsequent calls to the *gethostent()*, *gethostbyaddr()*, or *gethostbyname()* functions.
3. A coded character set identifier (CCSID) of 65535 requests that no database translation be performed. For translation to occur for the host names in the hostent structure, the job CCSID must be something other than 65535.
4. Do not use the *gethostent()* function in a multithreaded environment. See the multithread alternative *gethostent_r()* function.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *gethostent()* API is mapped to *qso_gethostent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “gethostbyaddr()—Get Host Information for IP Address” on page 222—Get Host Information for IP Address
- “gethostbyname()—Get Host Information for Host Name” on page 227—Get Host Information for Host Name
- “endhostent()—Close Host Database” on page 206—Close Host Database
- “sethostent()—Open Host Database” on page 325—Open Host Database

API introduced: V3R1

gethostent_r()—Get Next Entry from Host Database

Syntax

```
#include <netdb.h>
int gethostent_r(struct hostent
                    *hostent_struct_addr,
                 struct hostent_data
                    *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *gethostent_r()* function is used to retrieve information from the host database file. When the *gethostent_r()* is first called, the file is opened, and the first entry is returned. Each subsequent call of *gethostent_r()* results in the next entry in the file being returned. To close the file, use *endhostent_r()*.

Parameters

struct hostent *hostent_struct_addr (input/output)
Specifies the pointer to a hostent structure where the results will be placed. All results must be referenced through this structure.

struct hostent_data *hostent_data_struct_addr (input/output)
Specifies the pointer to the hostent_data structure, which is used to pass and preserve results between function calls. The field host_control_blk in the hostent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire hostent_data structure must be initialized to hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *gethostent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct hostent** denoted by ***hostent_struct_addr*** and **struct hostent_data**denoted by ***hostent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct hostent**is defined as:

```
struct hostent [
  char   *h_name;
  char   **h_aliases;
  int    h_addrtype;
  int    h_length;
  char   **h_addr_list;
];

#define h_addr  h_addr_list[0]
```

h_name points to the character string that contains the name of the host. *h_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the host. *h_addrtype* contains the address type of the host (for example, `af_inet`). *h_length*

contains the size of an address in octets (for example, the size of an Internet address is 4 octets). *h_addr_list* is a pointer to a NULL-terminated list of pointers, each of which points to a network address (in network byte order) for the host.

Error Conditions

When the *gethostent_r()* function fails, *errno* can be set to:

[EINVAL]

The hostent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure hostent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:

   - ADDTCPHTE (Add TCP/IP Host Table Entry)
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the host names returned in the hostent structure, the job CCSID must be something other than 65535.

Related Information

- "gethostbyaddr_r()—Get Host Information for IP Address" on page 224—Get Host Information for IP Address
- "gethostbyname_r()—Get Host Information for Host Name" on page 230—Get Host Information for Host Name
- "endhostent_r()—Close Host Database" on page 207—Close Host Database
- "sethostent_r()—Open Host Database" on page 326—Open Host Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getnameinfo()—Get Name Information for Socket Address

Syntax

```
#include  <sys/socket.h>
#include  <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *nodename, socklen_t nodenamelen,
                char *servname, socklen_t servnamelen,
                int flags);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***getnameinfo()*** function translates a socket address to a node name and service location, all of which are defined as with "getaddrinfo()—Get Address Information" on page 217.

Parameters

sa (Input) The pointer to a socket address structure to be translated.

salen (Input) The length of the socket address structure pointed to by *sa*.

nodename
(Output) If the *nodename* parameter is non-NULL and the *nodenamelen* parameter is nonzero, then the *nodename* parameter must point to a buffer able to contain up to *nodenamelen* characters that will receive the node name as a null-terminated string. If the *nodename* parameter is NULL or the *nodenamelen* parameter is zero, the node name will not be returned. If the node's name cannot be located, the numeric form of the nodes address is returned instead of its name.

nodenamelen
(Input) The length of the buffer pointed to by *nodename*

servname
(Output) If the *servname* parameter is non-NULL and the *servnamelen* parameter is nonzero, then the *servname* parameter must point to a buffer able to contain up to *servnamelen* characters that will receive the service name as a null-terminated string. If the *servname* parameter is NULL or the *servnamelen* parameter is zero, the service name will not be returned. If the service name cannot be located, the numeric form of the service address (for example, its port number) is returned instead of its name.

servnamelen
(Input) The length of the buffer pointed to by *servname*

flags (Input) A flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host is returned, unless one of the following is true:

- If the flag bit NI_NOFQDN is set, only the nodename portion of the FQDN is returned for local hosts.
- If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name, under all circumstances.
- If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.
- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name, under all circumstances.
- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior is to assume that the service is a stream service (SOCK_STREAM).

Authorities

No authorization required.

Return Value

getnameinfo() returns an integer. Possible values are:

- 0 (successful)
- non-zero (unsuccessful)

On successful completion, function ***getnameinfo()*** returns the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings, and may be truncated if the actual values are longer than can be stored in the buffers provided. If the returned values are to be used as part of any further name resolution (for example, passed to "getaddrinfo()—Get Address Information" on page 217

on page 217, callers must either provide buffers large enough to store any result possible on the system or must check for truncation and handle that case appropriately.

Error Conditions

When *getnameinfo()* fails, the error return value can be set to one of the following:

[EAI_AGAIN]

The name could not be resolved at this time. Future attempts may succeed.

[EAI_BADFLAGS]

The flags parameter had an invalid value.

[EAI_FAIL]

A non-recoverable error occurred.

[EAI_FAMILY]

The address family was not recognized or the address length was invalid for the specified family.

[EAI_MEMORY]

There was a memory allocation failure.

[EAI_NONAME]

The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were null.

[EAI_SYSTEM]

A system error occurred; the error code can be found in *errno*

Usage Notes

1. The *nodename* and *servname* parameters cannot both be NULL.
2. The "gai_strerror()—Retrieve Address Information Runtime Error Message" on page 216 API may be used to retrieve an error message associated with one of the error return values described above.
3. If the node and service information is obtained from the domain name server, the information is returned in the default coded character set identifier (CCSID) currently in effect for the job. (The default CCSID is the same as the job CCSID unless 65535 is requested, in which case the default CCSID is set based on the language ID of the job. See Globalization for more information.) If the node and service information is retrieved from the host database file, the default CCSID of the job is not used. To request conversion of the host information when it is retrieved from the host database file, you must use a job CCSID of something other than 65535.
4. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getnameinfo()* API is mapped to *qetnameinfo98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getaddrinfo()—Get Address Information" on page 217—Get Address Information
- "gai_strerror()—Retrieve Address Information Runtime Error Message" on page 216—Retrieve Address Information Runtime Error Message
- "gethostbyaddr()—Get Host Information for IP Address" on page 222—Get Host Information for IP Address
- "getservbyport()—Get Service Name for Port Number" on page 261—Get Service Name for Port Number

- "inet_ntop()—Convert IPv4 and IPv6 Addresses Between Binary and Text Form" on page 278—Convert IPv4 and IPv6 Addresses Between Binary and Text Form

API introduced: V5R2

---

getnetbyaddr()—Get Network Information for IP Address

BSD 4.3 Syntax

```
#include <netdb.h>

struct netent *getnetbyaddr(long network_address,
                            int address_type)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 240.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t network_address,
                            int address_type)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 240.

The *getnetbyaddr()* function is used to retrieve information about a network. The information is retrieved from the network database file.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

network_address
(Input) The 32-bit network IP address for which information is to be retrieved.

address_type
(Input) An integer that indicates the type of *network_address*.

Authorities

No authorization is required.

Return Value

getnetbyaddr() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct netent**.

The structure **struct netent** is defined in **<netdb.h>**.

```
struct netent {
  char            *n_name;
  char            **n_aliases;
  int             n_addrtype;
  unsigned long   n_net;
};
```

n_name points to the character string that contains the name of the network. *n_aliases* is a pointer to a NULL-terminated array of alternate names for the network. *n_addrtype* contains the address type of the network. *n_net* is the 32-bit network address (an IP address with host part set to zero).

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:
   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. The pointer returned by *getnetbyaddr()* points to static storage that is overwritten on subsequent calls to the *getnetbyaddr()*, *getnetbyname()*, or *getnetent()* functions.
3. When the network information is obtained from the network database file, the file is opened and the network information is retrieved (if it exists) from the file. The file is then closed only if a *setnetent()* with a nonzero parameter value was not previously done.
4. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the network names returned in the netent structure, the job CCSID must be something other than 65535.
5. Do not use the *getnetbyaddr()* function in a multithreaded environment. See the multithread alternative *getnetbyaddr_r()* function.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getnetbyaddr()* API is mapped to *qso_getnetbyaddr98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getnetbyname()—Get Network Information for Domain Name” on page 242—Get Network Information for Domain Name
- “getnetent()—Get Next Entry from Network Database” on page 245—Get Next Entry from Network Database
- “setnetent()—Open Network Database” on page 327—Open Network Database
- “endnetent()—Close Network Database” on page 209—Close Network Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getnetbyaddr_r()—Get Network Information for IP Address

Syntax

```
#include <netdb.h>

int getnetbyaddr_r(long network_address,
                   int address_type,
                   struct netent *netent_struct_addr,
                   struct netent_data
                      *netent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

A program uses the *getnetbyaddr_r()* function to retrieve information about a network. The information is retrieved from the network database file.

Parameters

long network_address (input)
Specifies the 32-bit network IP address for which information is to be retrieved.

int address_type (input)
Specifies an integer that indicates the type of *network_address*.

struct netent *netent_struct_addr (input/output)
Specifies the pointer to a netent structure where the results will be placed. All results must be referenced through this structure.

struct netent_data *netent_data_struct_addr (input/output)
Specifies the pointer to the netent_data structure, which is used to pass and preserve results between function calls. The field net_control_blk in the netent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire netent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getnetbyaddr_r()* function returns a integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct netent** denoted by ***netent_struct_addr*** and **struct netent_data**denoted by ***netent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct netent**is defined as:

```
struct netent [
  char            *n_name;
  char            **n_aliases;
  int             n_addrtype;
  unsigned long   n_net;
];
```

n_name points to the character string that contains the name of the network. *n_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the network. *n_addrtype* contains the address type of the network (that is, AF_INET). *n_net* is the 32-bit network address (that is, an IP address in network byte order with host part set to zero).

Error Conditions

When the *getnetbyaddr_r()* function fails, *errno* can be set to:

[EINVAL]

The netent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure netent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:

   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. When the network information is obtained from the network database file, the file is opened and the network information is retrieved (if it exists) from the file. The file is then closed only if a *setnetent_r()* call with a non-zero parameter value was not previously done.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the network names returned in the netent structure, the job CCSID must be something other than 65535.

Related Information

- "getnetent_r()—Get Next Entry from Network Database" on page 246-Get Next Entry from Network Database
- "getnetbyname_r()—Get Network Information for Domain Name" on page 244—Get Network Information for Domain Name
- "setnetent_r()—Open Network Database" on page 328-Open Network Database
- "endnetent_r()—Close Network Database" on page 210—Close Network Database

API introduced: V3R1

---

getnetbyname()—Get Network Information for Domain Name

BSD 4.3 Syntax

```
#include <netdb.h>

struct netent *getnetbyname(char *network_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 243.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct netent *getnetbyname(const char *network_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 243.

The *getnetbyname()* function is used to retrieve information about a network. The information is retrieved from the network database file.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

network_name
(Input) The pointer to the character string that contains the name of the network for which information is to be retrieved.

Authorities

No authorization is required.

Return Value

getnetbyname() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct netent**.

The structure **struct netent** is defined in **<netdb.h>**.

```
struct netent {
  char            *n_name;
  char            **n_aliases;
  int             n_addrtype;
  unsigned long   n_net;
};
```

n_name points to the character string that contains the name of the network. *n_aliases* is a pointer to a NULL-terminated array of alternate names for the network. *n_addrtype* contains the address type of the network. *n_net* is the 32-bit network address (an IP address with host part set to zero).

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:
   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. The pointer returned by *getnetbyname()* points to static storage that is overwritten on subsequent calls to the *getnetbyname()*, *getnetbyaddr()*, or *getnetent()* functions.
3. When the network information is obtained from the network database file, the file is opened and the network information is retrieved (if it exists) from the file. The file is then closed only if a *setnetent()* with a nonzero parameter value was not previously done.
4. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the network name specified by the *network_name* parameter, and for the network names returned in the netent structure, the job CCSID must be something other than 65535.
5. Do not use the *getnetbyname()* function in a multithreaded environment. See the multithread alternative *getnetbyname_r()* function.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getnetbyname()* API is mapped to *qso_getnetbyname98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getnetbyaddr()—Get Network Information for IP Address” on page 239—Get Network Information for IP Address
- “getnetent()—Get Next Entry from Network Database” on page 245—Get Next Entry from Network Database

- “setnetent()—Open Network Database” on page 327—Open Network Database
- “endnetent()—Close Network Database” on page 209—Close Network Database

API introduced: V3R1

---

getnetbyname_r()—Get Network Information for Domain Name

Syntax

```
#include <netdb.h>
int getnetbyname_r(char *network_name,
                   struct netent *netent_struct_addr,
                   struct netent_data
                      *netent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getnetbyname_r()* function is used to retrieve information about a network. The information is retrieved from the network database file.

Parameters

char *network_name (input/output)
Specifies the pointer to the character string that contains the name of the network for which information is to be retrieved.

struct netent *netent_struct_addr (input/output)
Specifies the pointer to a netent structure where the results will be placed. All results must be referenced through this structure.

struct netent_data *netent_data_struct_addr (input/output)
Specifies the pointer to the netent_data structure, which is used to pass and preserve results between function calls. The field net_control_blk in the netent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire netent_data structure must initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getnetbyname_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct netent** denoted by ***netent_struct_addr*** and **struct netent_data**denoted by ***netent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct netent**is defined as:

```
struct netent [
  char            *n_name;
  char            **n_aliases;
  int             n_addrtype;
  unsigned long   n_net;
];
```

n_name points to the character string that contains the name of the network. *n_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the network. *n_addrtype* contains the address type of the network (that is, AF_INET). *n_net* is the 32-bit network address (that is, an IP address in network byte order with host part set to zero).

Error Conditions

When the *getnetbyname_r()* function fails, *errno* can be set to:

[EINVAL]

The netent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure netent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:

   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. When the network information is obtained from the network database file, the file is opened and the network information is retrieved (if it exists) from the file. The file is then closed only if a *setnetent_r()* call with a non-zero parameter value was not previously done.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the network name specified by the *network_name* parameter, and for the network names returned in the netent structure, the job CCSID must be something other than 65535.

Related Information

- "getnetent_r()—Get Next Entry from Network Database" on page 246—Get Next Entry from Network Database
- "getnetbyaddr_r()—Get Network Information for IP Address" on page 240—Get Network Information for IP Address
- "setnetent_r()—Open Network Database" on page 328—Open Network Database
- "endnetent_r()—Close Network Database" on page 210—Close Network Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getnetent()—Get Next Entry from Network Database

Syntax

```
#include <netdb.h>

struct netent *getnetent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 246.

The *getnetent()* function is used to retrieve network information from the network database file. When *getnetent()* is first called, the file is opened, and the first entry is returned. Each subsequent call to *getnetent()* results in the next entry in the file being returned. To close the file, use *endnetent()*.

Authorities

No authorization is required.

Return Value

getnetent() returns a pointer. Possible values are:

- NULL (unsuccessful or end-of-file)
- p (successful), where p is a pointer to **struct netent**.

The structure **struct netent** is defined in **<netdb.h>**.

```
struct netent {
  char            *n_name;
  char            **n_aliases;
  int             n_addrtype;
  unsigned long   n_net;
};
```

n_name points to the character string that contains the name of the network. *n_aliases* is a pointer to a NULL-terminated array of alternate names for the network. *n_addrtype* contains the address type of the network. *n_net* is the 32-bit network address (an IP address with host part set to zero).

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:
   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. The pointer returned by *getnetent()* points to static storage that is overwritten on subsequent calls to the *getnetent()*, *getnetbyaddr()*, or *getnetbyname()* functions.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the network names returned in the netent structure, the job CCSID must be something other than 65535.
4. Do not use the *getnetent()* function in a multithreaded environment. See the multithread alternative *getnetent_r()* function.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getnetent()* API is mapped to *qso_getnetent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getnetbyaddr()—Get Network Information for IP Address” on page 239—Get Network Information for IP Address
- “getnetbyname()—Get Network Information for Domain Name” on page 242—Get Network Information for Domain Name
- “endnetent()—Close Network Database” on page 209—Close Network Database
- “setnetent()—Open Network Database” on page 327—Open Network Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getnetent_r()—Get Next Entry from Network Database

Syntax

```
#include <netdb.h>
int getnetent_r(struct netent *netent_struct_addr,
                struct netent_data
                   *netent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getnetent_r()* function is used to retrieve network information from the network database file. When the *getnetent_r()* is first called, the file is opened, and the first entry is returned. Each subsequent call of *getnetent_r()* results in the next entry in the file being returned. To close the file, use *endnetent_r()*.

Parameters

struct netent *_netent_struct_addr_ (input/output)
Specifies the pointer to a netent structure where the results will be placed. All results must be referenced through this structure.

struct netent_data *_netent_data_struct_addr_ (input/output)
Specifies the pointer to the netent_data structure, which is used to pass and preserve results between function calls. The field net_control_blk in the netent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire netent_data structure must initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getnetent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct netent**, denoted by ***netent_struct_addr*** and **struct netent_data**denoted by ***netent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct netent**is defined as:

```
struct netent [
  char            *n_name;
  char            **n_aliases;
  int             n_addrtype;
  unsigned long   n_net;
];
```

n_name points to the character string that contains the name of the network. *n_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the network. *n_addrtype* contains the address type of the network (that is, AF_INET). *n_net* is the 32-bit network address (that is, an IP address in network byte order with host part set to zero).

Error Conditions

When the *getnetent_r()* function fails, *errno* can be set to:

[EINVAL]

The netent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure netent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:

   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the network names returned in the netent structure, the job CCSID must be something other than 65535.

Related Information

- "getnetbyaddr_r()—Get Network Information for IP Address" on page 240—Get Network Information for IP Address
- "getnetbyname_r()—Get Network Information for Domain Name" on page 244—Get Network Information for Domain Name
- "setnetent_r()—Open Network Database" on page 328—Open Network Database
- "endnetent_r()—Close Network Database" on page 210—Close Network Database

API introduced: V3R1

---

getprotobyname()—Get Protocol Information for Protocol Name

BSD 4.3 Syntax

```
#include <netdb.h>

struct protoent *getprotobyname(char *protocol_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 249.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct protoent *getprotobyname(const char *protocol_name)
```

Service Program Name: QSOSRV2
Default Public Authority:*USE
Threadsafe: No; see "Usage Notes" on page 249.

The *getprotobyname()* function is used to retrieve information about a protocol. The information is retrieved from the protocol database file.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

protocol_name
(Input) The pointer to the character string that contains the name of the protocol for which information is to be retrieved.

Authorities

No authorization is required.

Return Value

getprotobyname() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct protoent**

The structure **struct protoent** is defined in **<netdb.h>**.

```
struct protoent {
  char            *p_name;
  char            **p_aliases;
  int             p_proto;
};
```

p_name points to the character string that contains the name of the protocol. *p_aliases* is a pointer to a NULL-terminated array of alternate names for the protocol. *p_proto* is the protocol number.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
2. The pointer returned by *getprotobyname()* points to static storage that is overwritten on subsequent calls to the *getprotobyname()*, *getprotobynumber()*, or *getprotoent()* functions.
3. When the protocol information is obtained from the protocol database file, the file is opened and the protocol information is retrieved (if it exists) from the file. The file is then closed only if a *setprotoent()* with a nonzero parameter value was not previously done.
4. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol name specified by the *protocol_name* parameter, and for the protocol names returned in the protoent structure, the job CCSID must be something other than 65535.
5. Do not use the *getprotobyname()* function in a multithreaded environment. See the multithread alternative *getprotobyname_r()* function.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getprotobyname()* API is mapped to *qso_getprotobyname98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getprotobynumber()—Get Protocol Information for Protocol Number” on page 251—Get Protocol Information for Protocol Number
- “getprotoent()—Get Next Entry from Protocol Database” on page 254—Get Next Entry from Protocol Database

- "setprotoent()—Open Protocol Database" on page 329—Open Protocol Database
- "endprotoent()—Close Protocol Database" on page 211—Close Protocol Database

API introduced: V4R2

---

getprotobyname_r()—Get Protocol Information for Protocol Name

Syntax

```
#include <netdb.h>
int getprotobyname_r(char *protocol_name,
                struct  protoent
                   *protoent_struct_addr,
                struct  protoent_data
                   *protoent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getprotobyname_r()* function is used to retrieve information about a protocol. The information is retrieved from the protocol database file.

Parameters

char *protocol_name (input)
Specifies the pointer to the character string that contains the name of the protocol for which information is to be retrieved.

struct protoent *protoent_struct_addr (input/output)
Specifies the pointer to a protoent structure where the results will be placed. All results must be referenced through this structure.

struct protoent_data *protoent_data_struct_addr (input/output)
Specifies the pointer to the protoent_data structure, which is used to pass and preserve results between function calls. The field proto_control_blk in the protoent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire protoent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getprotobyname_r()* returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct protoent** denoted by ***protoent_struct_addr*** and **struct protoent_data** denoted by ***protoent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct protoent**is defined as:

```
struct protoent [
  char            *p_name;
  char            **p_aliases;
  int             p_proto;
];
```

p_name points to the character string that contains the name of the protocol. *p_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the protocol. *p_proto* is the protocol number.

Error Conditions

When the *getprotobyname_r()* function fails, *errno* can be set to:

[EINVAL]

The protoent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure protoent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
2. When the protocol information is obtained from the protocol database file, the file is opened and the protocol information is retrieved (if it exists) from the file. The file is then closed only if a *setprotoent_r()* call with a non-zero parameter value was not previously done.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol name specified by the *protocol_name* parameter, and for the protocol names returned in the protoent structure, the job CCSID must be something other than 65535.

Related Information

- "getprotobynumber_r()—Get Protocol Information for Protocol Number" on page 253)—Get Protocol
- "getprotoent_r()—Get Next Entry from Protocol Database" on page 255—Get Next Entry from Protocol Database
- "setprotoent_r()—Open Protocol Database" on page 330—Open Protocol Database
- "endprotoent_r()—Close Protocol Database" on page 212—Close Protocol Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getprotobynumber()—Get Protocol Information for Protocol Number

Syntax

```
#include <netdb.h>

struct protoent
       *getprotobynumber(int protocol_number)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 252.

The *getprotobynumber()* function is used to retrieve information about a protocol. The information is retrieved from the protocol database file.

Parameters

protocol_number
(Input) The protocol number for which information is to be retrieved.

Authorities

No authorization is required.

Return Value

getprotobynumber() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct protoent**.

The structure **struct protoent** is defined in **<netdb.h>**.

```
struct protoent {
  char            *p_name;
  char            **p_aliases;
  int             p_proto;
};
```

p_name points to the character string that contains the name of the protocol. *p_aliases* is a pointer to a NULL-terminated array of alternate names for the protocol. *p_proto* is the protocol number.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
2. The pointer returned by *getprotobynumber()* points to static storage that is overwritten on subsequent calls to the *getprotobynumber()*, *getprotobyname()*, or *getprotoent()* functions.
3. When the protocol information is obtained from the protocol database file, the file is opened and the protocol information is retrieved (if it exists) from the file. The file is then closed only if a *setprotoent()* with a nonzero parameter value was not previously done.
4. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol names returned in the protoent structure, the job CCSID must be something other than 65535.
5. Do not use the *getprotobynumber()* function in a multithreaded environment. See the multithread alternative *getprotobynumber()* function.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getprotobynumber()* API is mapped to *qso_getprotobynumber98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getprotobyname()—Get Protocol Information for Protocol Name” on page 248—Get Protocol Information for Protocol Name
- “getprotoent()—Get Next Entry from Protocol Database” on page 254—Get Next Entry from Protocol Database
- “setprotoent()—Open Protocol Database” on page 329—Open Protocol Database
- “endprotoent()—Close Protocol Database” on page 211—Close Protocol Database

API introduced: V4R2

getprotobynumber_r()—Get Protocol Information for Protocol Number

Syntax

```
#include <netdb.h>

int getprotobynumber_r(int protocol_number,
                  struct protoent
                     *protoent_struct_addr,
                  struct protoent_data
                     *protoent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getprotobynumber_r()* function is used to retrieve information about a protocol. The information is retrieved from the protocol database file.

Parameters

int protocol_number (input)
Specifies the protocol number for which information is to be retrieved.

struct protoent *protoent_struct_addr (input/output)
Specifies the pointer to a protoent structure where the results will be placed. All results must be referenced through this structure.

struct protoent_data *protoent_data_struct_addr (input/output)
Specifies the pointer to the protoent_data structure, which is used to pass and preserve results between function calls. The field proto_control_blk in the protoent_data structures must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire protoent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getprotobynumber_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct protoent** denoted by ***protoent_struct_addr*** and **struct protoent_data** denoted by ***protoent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct protoent**is defined as:

```
struct protoent [
  char            *p_name;
  char            **p_aliases;
  int             p_proto;
];
```

p_name points to the character string that contains the name of the protocol. *p_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the protocol. *p_proto* is the protocol number.

Error Conditions

When the *getprotobynumber_r()* function fails, *errno* can be set to:

[EINVAL]

The protoent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure protoent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)

2. When the protocol information is obtained from the protocol database file, the file is opened and the protocol information is retrieved (if it exists) from the file. The file is then closed only if a *setprotoent_r()* call with a non-zero parameter value was not previously done.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol names returned in the protoent structure, the job CCSID must be something other than 65535.

Related Information

- "getprotobyname_r()—Get Protocol Information for Protocol Name" on page 250—Get Protocol Information for Protocol Name
- "getprotoent_r()—Get Next Entry from Protocol Database" on page 255—Get Next Entry from Protocol Database
- "setprotoent_r()—Open Protocol Database" on page 330—Open Protocol Database
- "endprotoent_r()—Close Protocol Database" on page 212—Close Protocol Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getprotoent()—Get Next Entry from Protocol Database

Syntax

```
#include <netdb.h>

struct protoent *getprotoent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 255.

The *getprotoent()* function is used to retrieve protocol information from the protocol database file. When *getprotoent()* is first called, the file is opened, and the first entry is returned. Each subsequent call to *getprotoent()* results in the next entry in the file being returned. To close the file, use *endprotoent()*.

Authorities

No authorization is required.

Return Value

getprotoent() returns a pointer. Possible values are:

- NULL (unsuccessful or end-of-file)
- p (successful), where p is a pointer to **struct protoent**.

The structure **struct protoent** is defined in **<netdb.h>**.

```
struct protoent {
  char            *p_name;
  char            **p_aliases;
  int             p_proto;
};
```

p_name points to the character string that contains the name of the protocol. *p_aliases* is a pointer to a NULL-terminated array of alternate names for the protocol. *p_proto* is the protocol number.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
2. The pointer returned by *getprotoent()* points to static storage that is overwritten on subsequent calls to the *getprotoent()*, *getprotobynumber()*, or *getprotobyname()* functions.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol names returned in the protoent structure, the job CCSID must be something other than 65535.
4. Do not use the *getprotoent()* function in a multithreaded environment. See the multithread alternative *getprotoent_r()* function.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getprotoent()* API is mapped to *qso_getprotoent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getprotobyname()—Get Protocol Information for Protocol Name” on page 248—Get Protocol Information for Protocol Name
- “getprotobynumber()—Get Protocol Information for Protocol Number” on page 251—Get Protocol Information for Protocol Number
- “endprotoent()—Close Protocol Database” on page 211—Close Protocol Database
- “setprotoent()—Open Protocol Database” on page 329—Open Protocol Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

getprotoent_r()—Get Next Entry from Protocol Database

Syntax

```
#include <netdb.h>
int getprotoent_r(struct protoent
                    *protoent_struct_addr,
                  struct protoent_data
                    *protoent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getprotoent_r()* function is used to retrieve protocol information from the protocol database file. When the *getprotoent_r()* is first called, the file is opened, and the first entry is returned. Each subsequent call of *getprotoent_r()* results in the next entry in the file being returned. To close the file, use *endprotoent_r()*.

Parameters

struct protoent *protoent_address (input/output)

Specifies the pointer to a protoent structure where the results will be placed. All results must be referenced through this structure.

struct protoent_data *protoent_data_struct_addr (input/output)

Specifies the pointer to the protoent_data structure, which is used to pass and preserve results between function calls. The field proto_control_blk in the protoent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire protoent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getprotoent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct protoent** denoted by ***protoent_struct_addr*** and **struct protoent_data** denoted by ***protoent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct protoent**is defined as:

```
struct protoent [
  char            *p_name;
  char            **p_aliases;
  int             p_proto;
];
```

p_name points to the character string that contains the name of the protocol. *p_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the protocol. *p_proto* is the protocol number.

Error Conditions

When the *getprotoent_r()* function fails, *errno* can be set to:

[EINVAL]

The protoent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure protoent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)

2. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol names returned in the protoent structure, the job CCSID must be something other than 65535.

Related Information

- "getprotobynumber_r()—Get Protocol Information for Protocol Number" on page 253—Get Protocol
- "getprotobyname_r()—Get Protocol Information for Protocol Name" on page 250—Get Protocol Information for Protocol Name
- "setprotoent_r()—Open Protocol Database" on page 330—Open Protocol Database
- "endprotoent_r()—Close Protocol Database" on page 212—Close Protocol Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getservbyname()—Get Port Number for Service Name

BSD 4.3 Syntax

```
#include <netdb.h>

struct servent *getservbyname(char *service_name,
                              char *protocol_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 258.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct servent *getservbyname(const char *service_name,
                              const char *protocol_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 258.

The *getservbyname()* function is used to retrieve information about services (the protocol being used by the service and the port number assigned for the service). The information is retrieved from the service database file.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

service_name
(Input) The pointer to the character string that contains the name of the service for which information is to be retrieved (for example, telnet).

protocol_name
(Input) The pointer to the character string that contains the name of the protocol that further qualifies the search criteria. For example, if the *service_name* is telnet, and the *protocol_name* is tcp, then the call will return the telnet server that uses the TCP protocol. If this parameter is set to NULL, then the first telnet server is returned, regardless of the protocol used.

Authorities

No authorization is required.

Return Value

getservbyname() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct servent**.

The structure **struct servent** is defined in **<netdb.h>**.

```
struct servent {
  char            *s_name;
  char            **s_aliases;
  int             s_port;
  char            *s_proto
};
```

s_name points to the character string that contains the name of the service. *s_aliases* is a pointer to a NULL-terminated array of alternate names for the service. *s_port* is the port number assigned to the service. *s_proto* is the protocol being used by the service.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the services database file:
   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
2. The pointer returned by *getservbyname()* points to static storage that is overwritten on subsequent calls to the *getservbyname()*, *getservbyname()*, or *getservent()* functions.
3. When the service information is obtained from the service database file, the file is opened and the service information is retrieved (if it exists) from the file. The file is then closed only if a *setservent()* with a nonzero parameter value was not previously done.
4. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the service name and the protocol name, specified by the *service_name* and *protocol_name* parameters, respectively, and for the service names returned in the servent structure, the job CCSID must be something other than 65535.
5. Do not use the *getservbyname()* function in a multithreaded environment. See the multithread alternative *getservbyname_r()* function.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getservbyname()* API is mapped to *qso_getservbyname98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

- "getservbyport()—Get Service Name for Port Number" on page 261—Get Service Name for Port Number
- "getservent()—Get Next Entry from Service Database" on page 264—Get Next Entry from Service Database
- "setservent()—Open Service Database" on page 331—Open Service Database
- "endservent()—Close Service Database" on page 213—Close Service Database

API introduced: V4R2

---

getservbyname_r()—Get Port Number for Service Name

Syntax

```
#include <netdb.h>
int getservbyname_r(char *service_name,
                    char *protocol_name,
                    struct servent
                       *servent_struct_addr,
                    struct servent_data
                       *servent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getservbyname_r()* function is used to retrieve information about services: the protocol being used by the service and the port number assigned for the service. The information is retrieved from the service database file.

Parameters

char *service_name (input)
Specifies the pointer to the character string that contains the name of the service for which information is to be retrieved (for example, telnet).

char *protocol_name (input)
Specifies the pointer to the character string that contains the name of the protocol that further qualifies the search search criteria. For example, if the *service_name* is telnet, and the *protocol_name* is tcp, then the call will return the telnet server that uses the TCP protocol. If this parameter is set to NULL, then the first telnet server is returned, regardless of the protocol used.

struct servent *servent_struct_addr (input/output)
Specifies the pointer to a servent structure where the results will be placed. All results must be referenced through this structure.

struct servent_data *servent_data_struct_addr (input/output)
Specifies the pointer to the servent_data structure, which is used to pass and preserve results between function calls. The field serve_control_blk in the servent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire servent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getservbyname_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct servent** denoted by ***servent_struct_addr*** and **struct servent_data**denoted by ***servent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct servent**is defined as:

```
struct servent [
  char            *s_name;
  char            **s_aliases;
  int             s_port;
  char            *s_proto
];
```

s_name points to the character string that contains the name of the service. *s_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the service. *s_port* is the port number assigned to the service. *s_proto* is a pointer to a character string that contains the name of the protocol being used by the service.

Error Conditions

When the *getservbyname_r()* function fails, *errno* can be set to:

[EINVAL]

?The servent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure servent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the services database file:

   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
2. When the service information is obtained from the service database file, the file is opened and the service information is retrieved (if it exists) from the file. The file is then closed only if a *setservent_r()* call with a non-zero parameter value was not previously done.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the following, the job CCSID must be something other than 65535:

   - The service name and the protocol name, specified by the *service_name* and *protocol_name* parameters, respectively
   - The service names returned in the servent structure

Related Information

- “getservbyport_r()—Get Service Name for Port Number” on page 262—Get Service Name for Port Number
- “getservent_r()—Get Next Entry from Service Database” on page 265—Get Next Entry from Service Database
- “setservent_r()—Open Service Database” on page 332—Open Service Database
- “endservent_r()—Close Service Database” on page 214—Close Service Database

API introduced: V3R1

getservbyport()—Get Service Name for Port Number

BSD 4.3 Syntax

```
#include <netdb.h>

struct servent *getservbyport(int port_number,
                              char *protocol_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 262.

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netdb.h>

struct servent *getservbyport(int port_number,
                              const char *protocol_name)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 262.

The *getservbyport()* function is used to retrieve information about a service assigned to a port number. The information is retrieved from the service database file.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

port_number
(Input) The port number for which service information is to be retrieved.

protocol_name
(Input) The pointer to the character string that contains the name of the protocol that further qualifies the search criteria. For example, if the *port_number* is 10, and the *protocol_name* is tcp, then the call will return the server that uses the TCP protocol on port number 10. If this parameter is set to NULL, then the first server is returned, regardless of the protocol used.

Authorities

No authorization is required.

Return Value

getservbyport() returns a pointer. Possible values are:

- NULL (unsuccessful)
- p (successful), where p is a pointer to **struct servent**.

The structure **struct servent** is defined in **<netdb.h>**.

```
struct servent {
  char            *s_name;
  char            **s_aliases;
  int             s_port;
  char            *s_proto
};
```

s_name points to the character string that contains the name of the service. *s_aliases* is a pointer to a NULL-terminated array of alternate names for the service. *s_port* is the port number assigned to the service. *s_proto* is the protocol being used by the service.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the services database file:

   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
2. The pointer returned by *getservbyport()* points to static storage that is overwritten on subsequent calls to the *getservbyport()*, *getservbyname()*, or *getservent()* functions.
3. When the service information is obtained from the service database file, the file is opened and the service information is retrieved (if it exists) from the file. The file is then closed only if a *setservent()* with a nonzero parameter value was not previously done.
4. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol name specified by the *protocol_name* parameter, or for the service names returned in the servent structure, the job CCSID must be something other than 65535.
5. Do not use the *getservbyport()* function in a multithreaded environment. See the multithread alternative *getservbyport_r()* function.
6. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getservbyport()* API is mapped to *qso_getservbyport98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getservbyname()—Get Port Number for Service Name” on page 257—Get Port Number for Service Name
- “getservent()—Get Next Entry from Service Database” on page 264—Get Next Entry from Service Database
- “setservent()—Open Service Database” on page 331—Open Service Database
- “endservent()—Close Service Database” on page 213—Close Service Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

getservbyport_r()—Get Service Name for Port Number

Syntax

```
#include <netdb.h>

int getservbyport_r(int port_number,
```

```
                char *protocol_name,
                struct servent *servent_struct_addr,
                struct servent_data
                   *servent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getservbyport_r()* function is used to retrieve information about a service assigned to a port number. The information is retrieved from the service database file.

Parameters

int port_number (input)
Specifies the port number for which service information is to be retrieved.

char *protocol_name (input)
Specifies the pointer to the character string that contains the name of the protocol that further qualifies the search criteria. For example, if the *port_number* is 10, and the *protocol_name* is tcp, then the call will return the server that uses the TCP protocol on port number 10. If this parameter is set to NULL, then the first server is returned, regardless of the protocol used.

struct servent *servent_struct_addr (input/output)
Specifies the pointer to a servent structure where the results will be placed. All results must be referenced through this structure.

struct servent_data *servent_data_struct_addr (input/output)
Specifies the pointer to the servent_data structure, which is used to pass and preserve results between function calls. The field serve_control_blk in the servent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required then the entire servent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getservbyport_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct servent** denoted by ***servent_struct_addr*** and **struct servent_data**denoted by ***servent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct servent**is defined as:

```
struct servent [
  char            *s_name;
  char            **s_aliases;
  int             s_port;
  char            *s_proto
];
```

s_name points to the character string that contains the name of the service. *s_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the service. *s_port* is the port number assigned to the service. *s_proto* is a pointer to a character string that contains the name of the protocol being used by the service.

Error Conditions

When the *getservbyport_r()* function fails, *errno* can be set to:

[EINVAL]

The servent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action see the description for structure servent_data.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the services database file:

   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
2. When the service information is obtained from the service database file, the file is opened and the service information is retrieved (if it exists) from the file. The file is then closed only if a *setservent_r()* call with a non-zero parameter value was not previously done.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the protocol name specified by the *protocol_name* parameter, or for the service names returned in the servent structure, the job CCSID must be something other than 65535.

Related Information

- "getservbyname_r()—Get Port Number for Service Name" on page 259—Get Port Number for Service Name
- "getservent_r()—Get Next Entry from Service Database" on page 265—Get Next Entry from Service Database
- "setservent_r()—Open Service Database" on page 332)—Open Service Database
- "endservent_r()—Close Service Database" on page 214—Close Service Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

getservent()—Get Next Entry from Service Database

Syntax

```
#include <netdb.h>

struct servent *getservent()
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 265.

The *getservent()* function is used to retrieve information about services (the protocol being used by the service and the port number assigned for the service). The information is retrieved from the services database file. When *getservent()* is first called, the file is opened, and the first entry is returned. Each subsequent call to *getservent()* results in the next entry in the file being returned. To close the file, use *endservent()*.

Authorities

No authorization is required.

Return Value

getservent() returns a pointer. Possible values are:

- NULL (unsuccessful or end-of-file)
- p (successful), where p is a pointer to **struct servent**.

```
struct servent {
  char            *s_name;
  char            **s_aliases;
  int             s_port;
  char            *s_proto
};
```

s_name points to the character string that contains the name of the service. *s_aliases* is a pointer to a NULL-terminated array of alternate names for the service. *s_port* is the port number assigned to the service. *s_proto* is the protocol being used by the service.

The structure **struct servent** is defined in **<netdb.h>**.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the services database file:
   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
2. The pointer returned by *getservent()* points to static storage that is overwritten on subsequent calls to the *getservent()*, *getservbyname()*, or *getservbyport()* functions.
3. A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the service names returned in the servent structure, the job CCSID must be something other than 65535.
4. Do not use the *getservent()* function in a multithreaded environment. See the multithread alternative *getservent_r()* function.
5. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *getservent()* API is mapped to *qso_getservent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “getservbyname()—Get Port Number for Service Name” on page 257—Get Port Number for Service Name
- “getservbyport()—Get Service Name for Port Number” on page 261—Get Service Name for Port Number
- “endservent()—Close Service Database” on page 213—Close Service Database
- “setservent()—Open Service Database” on page 331—Open Service Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

getservent_r()—Get Next Entry from Service Database

Syntax

```
#include <netdb.h>
int getservent_r(struct servent *servent_struct_addr,
                 struct servent_data
                    *servent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *getservent_r()* function is used to retrieve information about services: the protocol being used by the service and the port number assigned for the service. The information is retrieved from the services database file. When the *getservent_r()* is first called, the file is opened, and the first entry is returned. Each subsequent call of *getservent_r()* results in the next entry in the file being returned. To close the file, use *endservent_r()*.

Parameters

struct servent *servent_struct_addr (input/output)

Specifies the pointer to a servent structure where the results will be placed. All results must be referenced through this structure.

struct servent_data *servent_data_struct_addr (input/output)

Specifies the pointer to the servent_data structure, which is used to pass and preserve results between function calls. The field serve_control_blk in the servent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire servent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *getservent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct servent** denoted by ***servent_struct_addr*** and **struct servent_data**denoted by ***servent_data_struct_addr*** are both defined in **<netdb.h>**. The structure **struct servent**is defined as:

```
struct servent [
  char            *s_name;
  char            **s_aliases;
  int             s_port;
  char            *s_proto
];
```

s_name points to the character string that contains the name of the service. *s_aliases* is a pointer to a NULL-terminated list of pointers, each of which points to a character string that represents an alternative name for the service. *s_port* is the port number assigned to the service. *s_proto* is a pointer to a character string that contains the name of the protocol being used by the service.

Error Conditions

When the *getservent_r()* function fails, *errno* can be set to:

[einval]

The servent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure servent_data.

Usage Notes

The iSeries Navigator or the following CL commands can be used to access the services database file:

- WRKSRVTBLE (Work with Service Table Entries)
- ADDSRVTBLE (Add Service Table Entry)
- RMVSRVTBLE (Remove Service Table Entry)

A coded character set identifier (CCSID) of 65535 for the job requests that no database translation be performed. For translation to occur for the service names returned in the servent structure, the job CCSID must be something other than 65535.

Related Information

- "getservbyname_r()—Get Port Number for Service Name" on page 259—Get Port Number for Service Name
- "getservbyport_r()—Get Service Name for Port Number" on page 262—Get Service Name for Port Number
- "setservent_r()—Open Service Database" on page 332—Open Service Database
- "endservent_r()—Close Service Database" on page 214—Close Service Database

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

hstrerror()—Retrieve Resolver Error Message

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

char* hstrerror(int h_error_value);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The **hstrerror()** function is used to retrieve the text string that describes a resolver *h_errno* value.

Parameters

h_error_value (Input)
The *h_errno* received from a resolver API.

Return Value

The **hstrerror()** API returns a pointer to the error text.

Authorities:

No authorization is required.

Error Conditions

None

Usage Notes

1. If the *h_error_value* is out of range or not found, "Unknown resolver error" will be returned.

Related Information

- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nmkupdate()—Construct an Update Packet" on page 306—Construct an Update Packet
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_nsendsigned()—Send Authenticated Domain Query or Update" on page 311—Send Authenticated Domain Query
- "res_nupdate()—Build and Send Dynamic Updates" on page 314—Build and Send Dynamic Updates
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

Example

See Code disclaimer information for information pertaining to code examples.

See "res_ninit()—Initialize res Structure" on page 299 for an example of how *hstrerror()* is used.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

htonl()—Convert Long Integer to Network Byte Order

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long htonl(unsigned long host_long)
```

Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netinet/in.h>

uint32_t htonl(uint32_t host_long)
```

Threadsafe: Yes

The *htonl()* function is used to convert a long (4-byte) integer from the local host byte order to standard network byte order.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

host_long
(Input) The 4-byte integer in local host byte order that is to be converted to standard network byte order.

Authorities

No authorization is required.

Return Value

htonl() returns an integer. Possible values are:

- n (where n is the 4-byte integer in standard network byte order)

Usage Notes

1. On the iSeries server, the value returned to the caller is the same as the value that was passed to *htonl()*, since the local host byte order does not differ from the standard network byte order.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "ntohl()—Convert Long Integer to Host Byte Order" on page 285)—Convert Long Integer to Host Byte Order
- "htons()—Convert Short Integer to Network Byte Order"—Convert Short Integer to Network Byte Order
- "ntohs()—Convert Short Integer to Host Byte Order" on page 286—Convert Short Integer to Host Byte Order

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

htons()—Convert Short Integer to Network Byte Order

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned short htons(unsigned short host_short)
```

Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netinet/in.h>

uint16_t htons(uint16_t host_short)
```

Threadsafe: Yes

The *htons()* function is used to convert a short (2-byte) integer from the local host byte order to standard network byte order.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

host_short
(Input) The 2-byte integer in local host byte order that is to be converted to standard network byte order.

Authorities

No authorization is required.

Return Value

htons() returns an integer. Possible values are:

- n (where n is the 2-byte integer in standard network byte order)

Usage Notes

1. On the iSeries server, the value returned to the caller will be the same as the value that was passed to *htons()*, since the local host byte order does not differ from the standard network byte order.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "ntohs()—Convert Short Integer to Host Byte Order" on page 286—Convert Short Integer to Host Byte Order
- "htonl()—Convert Long Integer to Network Byte Order" on page 268—Convert Long Integer to Network Byte Order
- "ntohl()—Convert Long Integer to Host Byte Order" on page 285—Convert Long Integer to Host Byte Order

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

inet_addr()—Translate Full Address to 32-bit IP Address

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *address_string)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <arpa/inet.h>

in_addr_t inet_addr(const char *address_string)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *inet_addr()* function is used to translate an Internet address from dotted decimal format to a 32-bit IP address.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

address_string
(Input) The Internet address in dotted decimal format that is to be converted to a 32-bit IP address.

Authorities

No authorization is required.

Return Value

inet_addr() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (where n is the 32-bit IP address)

The *inet_addr()* subroutine returns an error value of -1 for strings that are not valid.

Note: An Internet address with a dot notation value of 255.255.255.255 or its equivalent in a different base format causes the *inet_addr()* subroutine to return an unsigned long value of 4294967295. This value is identical to the unsigned representation of the error value. Otherwise, the *inet_addr()* subroutine considers 255.255.255.255 a valid Internet address.

Error Conditions

When *inet_addr()* fails, *errno* can be set to one of the following:

[EFAULT]

Bad address.

The system detected an address which was not valid while attempting to access the *address_string* parameter.

[EINVAL]

Parameter not valid.

Usage Notes

1. Notation of the dotted decimal address string can be in one of seven formats:
   - Format 1 - a.b.c.d
   - Format 2 - a.b.c.
   - Format 3 - a.b.c
   - Format 4 - a.b.
   - Format 5 - a.b
   - Format 6 - a.

- Format 7 - a

Where a component of the dotted decimal format can be decimal (for example, 7.3), octal (for example, 07.3) or hexadecimal (for example, 0xb.3).

The rules for converting a dotted decimal string are as follows:

- For format 1, each component is interpreted as one byte of the internet address.
- For format 2, each component is interpreted as one byte of the internet address, and the rightmost byte is set to zero.
- For format 3, each component is interpreted as one byte of the internet address, except for component c, which is interpreted as the rightmost two bytes of the internet address.
- For format 4, each component is interpreted as one byte of the internet address, and the rightmost two bytes are set to zero.
- For format 5, each component is interpreted as one byte of the internet address, except for component b, which is interpreted as the rightmost three bytes of the internet address.
- For format 6, component a is interpreted as one byte of the internet address, and the rightmost three bytes are set to zero.
- For format 7, component a is returned as the internet address.

2. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *inet_addr()* API is mapped to *qso_inet_addr98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

API introduced: V3R1

---

inet_lnaof()—Separate Local Portion of IP Address

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_lnaof(struct in_addr internet_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <arpa/inet.h>

in_addr_t inet_lnaof(struct in_addr internet_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *inet_lnaof()* function is used to extract the local host portion of an IP address.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

internet_address
(Input) The 32-bit IP address from which the local host portion of the address is to be extracted.

Authorities

No authorization is required.

Return Value

inet_lnaof() returns an integer. Possible values are:

- n (where n is the local host address)

Usage Notes

1. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *inet_lnaof()* API is mapped to *qso_inet_lnaof98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "inet_makeaddr()—Combine Network Portion and Host Portion to Make IP Address"—Combine Network Portion and Host Portion to Make IP Address
- "inet_netof()—Separate Network Portion of IP Address" on page 274—Separate Network Portion of IP Address

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

inet_makeaddr()—Combine Network Portion and Host Portion to Make IP Address

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr(int network_address,
                             int host_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <arpa/inet.h>

struct in_addr inet_makeaddr(in_addr_t network_address,
                             in_addr_t host_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *inet_makeaddr()* function is used to generate a 32-bit IP address from the 32-bit network IP address and the local address of the host.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

network_address
(Input) The 32-bit network IP address.

host_address
(Input) The local host address.

Authorities

No authorization is required.

Return Value

inet_makeaddr() returns an integer. Possible values are:

- n (where n is the 32-bit IP address)

When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *inet_makeaddress()* API is mapped to *qso_inet_makeaddress98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

inet_netof()—Separate Network Portion of IP Address

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_netof(struct in_addr internet_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <arpa/inet.h>

in_addr_t inet_netof(struct in_addr internet_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *inet_netof()* function is used to extract the network portion of an IP address.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

internet_address
(Input) The 32-bit IP address from which the network portion of the address is to be extracted.

Authorities

No authorization is required.

Return Value

inet_netof() returns an integer. Possible values are:

- n (where n is the network IP address)

Usage Notes

1. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *inet_netof()* API is mapped to *qso_inet_netof98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- “inet_lnaof()—Separate Local Portion of IP Address” on page 272—Separate Local Portion of IP Address
- “inet_makeaddr()—Combine Network Portion and Host Portion to Make IP Address” on page 273—Combine Network Portion and Host Portion to Make IP Address

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

inet_network()—Translate Network Portion of Address to 32-bit IP Address

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network(char *address_string)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <arpa/inet.h>

in_addr_t inet_network(const char *address_string)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *inet_network()* function is used to translate an Internet address from dotted decimal format to a 32-bit network IP address, in which the host part of the IP address is set to zeros.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

address_string
(Input) The Internet address in dotted decimal format that is to be converted to a 32-bit network IP address.

Authorities

No authorization is required.

Return Value

inet_network() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (where n is the 32-bit network IP address)

Error Conditions

When *inet_network()* fails, *errno* can be set to one of the following:

[EFAULT]

Bad address.

The system detected an address which was not valid while attempting to access the *address_string* parameter.

[EINVAL]

Parameter not valid.

When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *inet_network()* API is mapped to *qso_inet_network98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

inet_ntoa()—Translate IP Address to Dotted Decimal Format

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr internet_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *inet_ntoa()* function is used to translate an Internet address from a 32-bit IP address to dotted decimal format.

Authorities and Locks

None.

Parameters

internet_address
(Input) The 32-bit IP address that is to be converted to dotted decimal format.

Return Value

inet_ntoa() returns one of the following values:

- NULL (unsuccessful)
- s (where s is the pointer to the Internet address in dotted decimal format)

Usage Notes

1. The pointer returned by *inet_ntoa()* points to static storage that is overridden on subsequent *inet_ntoa()* functions.
2. Do not use the *inet_ntoa()* function in a multithreaded environment. See the multithread alternative *inet_ntoa_r* function.

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

inet_ntoa_r()—Translate IP Address to Dotted Decimal Format

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int              inet_ntoa_r(struct in_addr internet_address,
             char *output_buffer,
             int  output_buffer_length)
```

Service Program Name: Name QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *inet_ntoa_r()* function is used to translate an Internet address from a 32-bit IP address to dotted decimal format.

Authorities and Locks

None.

Parameters

struct in_addr *internet_address* (input)
The 32-bit IP address that is to be converted to dotted decimal format.

char * *output_buffer* (input/output)
The pointer to the buffer that contains the dotted decimal format.

int *output_buffer_length* (input)
The length of the output buffer (length should be at least 16).

Return Value

The *inet_ntoa_r()* function returns:

- -1 (unsuccessful call)
- 0 (successful call)

Error Conditions

When the *inet_ntoa_r()* function fails, *errno* can be set to:

[EINVAL]
Parameter is not valid.

This error code indicates one of the following:

- The *output_buffer_length* length is less than 16.

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

inet_ntop()—Convert IPv4 and IPv6 Addresses Between Binary and Text Form

Syntax

```
#include  <sys/socket.h>
#include  <arpa/inet.h>

const char *inet_ntop(int af, const void *src,
                      char *dst, socklen_t size);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***inet_ntop()*** function converts a numeric address into a text string suitable for presentation.

Parameters

af (Input) Specifies the family of the address to be converted. Currently the AF_INET and AF_INET6 address families are supported.

src (Input) The pointer to a buffer that contains the numeric form of an IPv4 address if the *af* parameter is AF_INET, or the numeric form of an IPv6 address if the *af* parameter is AF_INET6.

dst (Output) The pointer to a a buffer into which the function stores the resulting null-terminated text string.

size (Input) The size of the buffer pointed at by *dst*. The calling application must ensure that the buffer referred to by *dst* is large enough to hold the resulting text string. For IPv4 addresses, the buffer must be at least 16 bytes. For IPv6 addresses, the buffer must be at least 46 bytes. In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in <netinet/in.h>:

```
#define INET_ADDRSTRLEN    16
#define INET6_ADDRSTRLEN   46
```

Authorities

No authorization is required.

Return Value

inet_ntop() returns a pointer. Possible values are:

- NULL (unsuccessful)
- non-NULL (successful)

If successful, ***inet_ntop()*** returns a pointer to the buffer containing the text string.

Error Conditions

When *inet_ntop()* fails, *errno* will be set to one of the following:

[EAFNOSUPPORT]

The address family is not supported.

[ENOSPC]

The size of the result buffer is inadequate.

[EINVAL]

Parameter is not valid.

[EFAULT]

The system detected an address which was not valid while attempting to access the *src* or *dst* parameter.

Usage Notes

1. The resulting string will be in the standard IPv4 dotted-decimal format for IPv4 or one of the preferred forms for IPv6. See the Usage Notes for “inet_pton()—Convert IPv4 and IPv6 Addresses Between Text and Binary Form” on page 280 for a more detailed description.

2. A job has a coded character set identifier (CCSID). The job CCSID will be used to convert the characters stored at *dst* (to allow the hexadecimal values to be shown in lower case).

Related Information

- "inet_ntoa()—Translate IP Address to Dotted Decimal Format" on page 277—Translate IP Address to Dotted Decimal Format
- "inet_pton()—Convert IPv4 and IPv6 Addresses Between Text and Binary Form"—Convert IPv4 and IPv6 Addresses Between Text and Binary Form

API introduced: V5R2

Top | UNIX-Type APIs | APIs by category

---

inet_pton()—Convert IPv4 and IPv6 Addresses Between Text and Binary Form

Syntax

```
#include  <sys/socket.h>
#include  <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The ***inet_pton()*** function converts an address in its standard text presentation form into its numeric binary form.

Parameters

af (Input) Specifies the family of the address to be converted. Currently the AF_INET and AF_INET6 address families are supported.

src (Input) The pointer to the null-terminated character string that contains the text presentation form of an IPv4 address if the *af* parameter is AF_INET, or the text presentation form of an IPv6 address if the *af* parameter is AF_INET6. See usage notes for the supported formats.

dst (Output) The pointer to a buffer into which the function stores the numeric address. The calling application must ensure that the buffer referred to by *dst* is large enough to hold the numeric address (4 bytes for AF_INET or 16 bytes for AF_INET6).

Authorities

No authorization is required.

Return Value

inet_pton() returns an integer. Possible values are:

- 1 (successful)
- 0 (unsuccessful—input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string)
- -1 (unsuccessful—see *errno*)

If successful, the buffer pointed at by *dst* will be updated with the numeric address.

Error Conditions

When *inet_pton()* fails with a -1, *errno* will be set to:

[EAFNOSUPPORT]

The address family is not supported.

[EINVAL]

Parameter is not valid.

[EFAULT]

The system detected an address which was not valid while attempting to access the *src* or *dst* parameter.

Usage Notes

1. If the *af* parameter of ***inet_pton()*** is AF_INET, the *src* string must be in the standard IPv4 dotted-decimal form:

   ddd.ddd.ddd.ddd

   where ddd is a one to three digit decimal number between 0 and 255 (see the "inet_addr()—Translate Full Address to 32-bit IP Address" on page 270 definition). The ***inet_pton*** function does not accept other formats (such as the octal numbers, hexadecimal numbers, and fewer than four numbers that "inet_addr()—Translate Full Address to 32-bit IP Address" on page 270 accepts).
2. If the *af* parameter of ***inet_pton*** is AF_INET6, the *src* string must be in one of the following IPv6 text forms:
   a. The preferred form is **x:x:x:x:x:x:x:x**, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there must be at least one value in every field.
   b. A string of contiguous zero fields in the preferred form can be shown as "::". The "::" can only appear once in an address. Unspecified addresses (**0:0:0:0:0:0:0:0**) may be represented simply as "::".
   c. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is **x:x:x:x:x:x:d.d.d.d**, where the "x"s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the "d"s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).
3. » The above IPv6 text forms may include an appended zone indicator (if preceded by a % character) and/or an appended prefix length (if preceded by a / character). In these cases, the % or / will be treated the same as a null terminator.
4. A trailing space will be treated the same as a null terminator.
5. The default coded character set identifier (CCSID) currently in effect for the job « will be used to convert the characters found at *src* (to allow the hexadecimal values to be entered in lower case).

Related Information

- "inet_addr()—Translate Full Address to 32-bit IP Address" on page 270—Translate Full Address to 32-bit IP Address
- "inet_ntop()—Convert IPv4 and IPv6 Addresses Between Binary and Text Form" on page 278—Convert IPv4 and IPv6 Addresses Between Binary and Text Form

API introduced: V5R2

ns_addr()—Translate Network Services Address to 12-byte Address

Syntax

```
#include <sys/types.h>
#include <netns/ns.h>

struct ns_addr ns_addr(char *address_string)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *ns_addr()* function is used to translate a network services address from human readable format to a 12-byte hexadecimal address.

Authorities and Locks

None.

Parameters

char ***address_string***
(Input) The network services address in human readable format.

Return Value

The *ns_addr()* function returns an **ns_addr** structure.

Usage Notes

Notation of the human readable address string can be in many forms. The following notation rules apply to all the format examples shown here.

1. There are three fields to the address string: the network field denoted by bytes n1 through n4, the host field denoted by bytes h1 through h6, and the port number field denoted by bytes p1 and p2. These three fields can be separated by a period (.), a colon (:), or a (#). Once one of these three separator characters is encountered, the rest of the fields (the host field and the port number field) may be byte separated by a period or a colon. The network field cannot use byte separators because it is the first field and a field separator has not been encountered. Also, you may not use the same character as a field separator and a byte separator.
2. Each field may be specified as either decimal, hexadecimal, or octal. Octal is specified by a preceding zero (for example, 011 is decimal value 9). Hexadecimal can be specified in the following ways:
   - Specifying 0xnn.
   - Specifying 0Xnn.
   - Specifying xnn.
   - Specifying Xnn.
   - Specifying an H character at the end of the field.
   - Using a byte separator (only allowed for the host field or port number) in the field that contains the byte.
   - Using any of the characters a,b,c,d,e,f,A,B,C,D,E,F in any byte in the field.

The following are valid formats:

- Format 1 - `n1n2n3n4:h1.h2.h3.h4.h5.h6:p1.p2`
- Format 2 - `n1n2n3n4.h1:h2:h3:h4:h5:h6.p1:p2`
- Format 3 - `n1n2n3n4#h1.h2.h3.h4.h5.h6#p1.p2`
- Format 4 - `n1n2n3n4#h1:h2:h3:h4:h5:h6#p1:p2`

Although they can have byte separators, the host and port fields do not need to be byte separated. Also, not all bytes need be specified for a given field. If not all bytes are specified, the specified bytes are right-justified in the field.

Note: If the host field is not byte separated, the number must not be larger than what can be contained in a 4-byte integer. That is, to use nonzero values for bytes h1 and h2, you must byte separate the host field.

The following formats are also valid:
- Format 5 - `n1n2n3n4:h1h2h3h4h5h6:p1p2`
- Format 6 - `n1:h1.h2.h3.h4.h5.h6:p1p2`
- Format 7 - `n1:h1h2h3h4h5h6:p1.p2`

Not all fields need be specified. The following formats are also valid:
- Format 8 - n1
- Format 9 - n1:h1
- Format 10 - n1::p1
- Format 11 - ::p1

As a further example, the following are just some of the ways that a network number of 71 decimal, a host number of 8374930 decimal, and a port number of 9341 can be specified.
- 71:8374930:9341
- 71:00.00.00.7f.ca.92:9341
- 71:7f.ca.92:9341
- 0x47:7fca92:247d
- 47H:7f.ca.92:9341
- 47H.7fca92.247d

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

ns_ntoa()—Translate Network Services Address from 12-byte Address/h2>

Syntax

```
#include <sys/types.h>
#include <netns/ns.h>

char *ns_ntoa
     (struct ns_addr network_services_address)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see “Usage Notes” on page 284.

The *ns_ntoa()* function is used to translate a network services address from a 12-byte address to a human readable format.

Authorities and Locks

None.

Parameters

struct ns_addr ***network_services_address***

(Input) The 12-byte network services address that is to be converted to human readable format.

Return Value

The *ns_ntoa()* function returns:

- NULL (unsuccessful call)
- s (where s is the pointer to the network services address in human readable format)

Usage Notes

1. The network services address consists of three fields, the network field, the host field, and the port number field. *ns_ntoa()* returns these fields as a single character string with the fields separated by the period (.) character. The character string is always terminated with a NULL character.
2. The fields are always returned in hexadecimal notation. *ns_ntoa()* inserts an H character at the end of each field that does not contain an a,b,c,d,e,f,A,B,C,D,E or F character, in order to make it obvious that the notation is in hexadecimal.
3. Not all fields need be returned. For example, if the host field and the port number field of the network services address both contain hexadecimal zeros, *ns_ntoa()* returns a character string that only contains the network field.
4. The pointer returned by *ns_ntoa()* points to static storage that is overridden on subsequent calls to *ns_ntoa()*.
5. Do not use the ns_ntoa() function in a multithread environment. See the multithread alternative "ns_ntoa_r() — Translate Network Services Address from 12-byte Address" function.

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

ns_ntoa_r() — Translate Network Services Address from 12-byte Address

Syntax

```
#include <sys/types.h>
#include <netns/ns.h>

int ns_ntoa_r(struct ns_addr
                *network_services_address,
              char *output_buffer,
              int  output_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *ns_ntoa_r()* function is used to translate a network services address from a 12-byte address to a human readable format.

Authorities and Locks

None.

Parameters

struct ns_addr network_services_address (input)
Specifies the 12-byte network services address that is to be converted to human readable format.

char * output_buffer (input/output)
Specifies the pointer to the converted string.

int output_buffer_length (input)
Specifies the length of the output buffer (length should at least 35).

Return Value

The *ns_ntoa_r()* function returns:

- -1 (unsuccessful call)
- 0 (successful call)

Error Conditions

When the *ns_ntoa_r()* function fails, *errno* can be set to:

[EINVAL]

Parameter is not valid.

This error code indicates one of the following:

- The *output_buffer_length* length is less than 35.

Usage Notes

1. The network services address consists of three fields, the network field, the host field, and the port number field. *ns_ntoa_r()* will return these fields as a single character string with the fields separated by the period (.) character. The character string is always terminated with a NULL character.
2. The fields are always returned in hexadecimal notation. *ns_ntoa_r()* will insert an 'H' character at the end of each field that does not contain an a,b,c,d,e,f,A,B,C,D,E or F character, in order to make it obvious that the notation is in hexadecimal.
3. Not all fields need be returned. For example, if the host field and the port number field of the network services address both contain hexadecimal zeros, the *ns_ntoa_r()* routine will return a character string that only contains the network field.

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

ntohl()—Convert Long Integer to Host Byte Order

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long ntohl(unsigned long network_long)
```

Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netinet/in.h>

uint32_t ntohl(uint32_t network_long)
```

Threadsafe: Yes

The *ntohl()* function is used to convert a long (4-byte) integer from the standard network byte order to the local host byte order.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

network_long
(Input) The 4-byte integer in standard network byte order that is to be converted to local host byte order.

Authorities

No authorization is required.

Return Value

ntohl() returns an integer. Possible values are:

- n (where n is the 4-byte integer in local host byte order)

Usage Notes

On the iSeries server, the value returned to the caller is the same as the value that was passed to *ntohl()*, since the standard network byte order does not differ from the local host byte order.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "htonl()—Convert Long Integer to Network Byte Order" on page 268—Convert Long Integer to Network Byte Order
- "htons()—Convert Short Integer to Network Byte Order" on page 269—Convert Short Integer to Network Byte Order
- "ntohs()—Convert Short Integer to Host Byte Order"—Convert Short Integer to Host Byte Order

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

ntohs()—Convert Short Integer to Host Byte Order

BSD 4.3 Syntax

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned short ntohs(unsigned short network_short)
```

Threadsafe: Yes

UNIX 98 Compatible Syntax

```
#define _XOPEN_SOURCE 520
#include <netinet/in.h>

uint16_t ntohs(uint16_t network_short)
```

Threadsafe: Yes

The *ntohs()* function is used to convert a short (2-byte) integer from the standard network byte order to the local host byte order.

There are two versions of the API, as shown above. The base i5/OS API uses BSD 4.3 structures and syntax. The other uses syntax and structures compatible with the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface with the _XOPEN_SOURCE macro.

Parameters

network_short
(Input) The 2-byte integer in standard network byte order that is to be converted to local host byte order.

Authorities

No authorization is required.

Return Value

ntohs() returns an integer. Possible values are:

- n (where n is the 2-byte integer in local host byte order)

Usage Notes

On the iSeries server, the value returned to the caller is the same as the value that was passed to *ntohs()*, since the standard network byte order does not differ from the local host byte order.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "htons()—Convert Short Integer to Network Byte Order" on page 269—Convert Short Integer to Network Byte Order
- "htonl()—Convert Long Integer to Network Byte Order" on page 268—Convert Long Integer to Network Byte Order
- "ntohl()—Convert Long Integer to Host Byte Order" on page 285—Convert Long Integer to Host Byte Order

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

res_close()—Close Socket and Reset _res Structure

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_close(void)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_close()* function is used to reset the **_res** structure to the beginning defaults and close a socket that is opened as a result of the RES_STAYOPEN flag.

Authorities:

No authorization is required.

Return Value

None

Usage Notes

1. If *res_send()* was previously called with RES_STAYOPEN set in the options field of the **_res** structure, *res_close()* closes the socket that was left open. *res_close()* does not attempt the close if there was no socket left open.
2. *res_close()* sets the **_res** structure to default values.
   - The *retrans* field is set to 5.
   - The *retry* field is set to 4.
   - The *options* field has the RES_RECURSE, RES_DEFDNAMES, and RES_DNSSRCH bits set.
   - The *nscount* field is set to 1.
   - All other fields in the **_res** structure are cleared.
   - In a thread-enabled environment **_res** structure is shared among all threads within a process.

Related Information

- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_mkquery()—Place Domain Query in Buffer" on page 296—Place Domain Query in Buffer
- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_search()—Search for Domain Name" on page 318—Search for Domain Name
- "res_send()—Send Buffered Domain Query or Update" on page 320—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

res_findzonecut()—Find the Enclosing Zone and Servers

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_findzonecut(state* res,
                    const char *domain_name,
                    ns_class class,
                    int options,
                    char *zone_name,
                    size_t zone_size,
                    struct in_addr *addresses,
                    int num_addresses)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_findzonecut()* queries name servers until it finds the enclosing zone and its master name servers for the specified domain name.

Parameters

res (Input) The pointer to the **state** structure.

domain_name
(Input) The pointer to the domain name whose enclosing zone is desired.

class (Input) The class of *domain_name*.

options
(Input) Processing options, may be RES_EXHAUSTIVE.

zone_name
(Output) The pointer to the enclosing zone name found.

zonesize
(Input) The size of the zone_name buffer.

addresses
(Output) The name server addresses found for the enclosing zone.

num_addresses
(Input) The maximum number of addresses to be returned.

Authorities

No authorization is required.

Return Value

res_findzonecut() returns an integer. Possible values are:

- < 0 - (unsuccessful).
- = 0 - *zone_name* is now valid, but *addresses* wasn't changed.
- > 0 - *zone_name* is now valid, and the return value is number of *addresses* found.

Error Conditions

When the *res_findzonecut()* function fails, *res_findzonecut()* can set *errno* to one of the following:

[ECONVERT]

Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

[EDESTADDRREQ]

No zone could be found for the domain.

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

One of the following reasons:

- An invalid length or NULL pointer was passed to *res_findzonecut()*
- The **res** appears to be initialized but the reserved field is not set to zeros.

Note: No attempt is made to initialize the **res** structure if it was initialized previous to the *res_findzonecut()* being issued.

[EMSGSIZE]

An invalid message length was returned on an answer.

[EPROTOTYPE]

The answer to a query had the wrong domain name.

Note: There are numerous other values that *errno* can be set to by the resolver and sockets functions that *res_findzonecut()* calls. Refer to other functions for the other values.

Usage Notes

1. *res_findzonecut()* calls *res_mkquery()* and *res_send()* to query the specified server for the zone information.
2. *res_findzonecut()* calls *res_ninit()* if the **res** structure has not been initialized.
3. *res_findzonecut()* assumes that the data passed to it is EBCDIC and is in the default coded character set identifier (CCSID) currently in effect for the job. It translates the data from the default CCSID currently in effect for the job to ASCII (CCSID 819) before the data is sent out to a name server. The response that it receives from the name server is returned in the default CCSID currently in effect for the job.

Related Information

- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nmkupdate()—Construct an Update Packet" on page 306—Construct an Update Packet
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_nsendsigned()—Send Authenticated Domain Query or Update" on page 311—Send Authenticated Domain Query
- "res_nupdate()—Build and Send Dynamic Updates" on page 314—Build and Send Dynamic Updates
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V5R1

res_hostalias()—Retrieve the host alias

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

const char * res_hostalias(const state* res,
                           const char* name,
                           char* destination,
                           size_t destination_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_hostalias()* looks up the specified name in the host aliases file specified by the environment variable *HOSTALIASES*.

A user may create a host aliases file. This file maps user defined aliases to host names, unlike the i5/OS host table (or a DNS) which maps host names to ip addresses. Also, it requires no special authorities for a user to define an alias. It's simply a shorthand for a server which can be easily changed and controlled by users. No iSeries server default alias file is created.

The format is simply an alias followed by blank(s) followed by a domain name. For example, mypc may be an alias for m999.mydomain.ibm.com and myaix may be an alias for m111.mydomain.ibm.com:

mypc m999.mydomain.ibm.com.

myaix m111.mydomain.ibm.com

Other functions, like "res_nsearch()—Search for Domain Name" on page 309 or "gethostbyname_r()—Get Host Information for Host Name" on page 230 will resolve an alias like ″mypc″ to the full domain name ″m999.mydomain.ibm.com.″ before querying the DNS or i5/OS host table.

Note: An alias may not contain periods.

Parameters

res (Input) The pointer to the **state** structure.

name (Input) The pointer to the host name.

destination
(Output) The pointer to the destination buffer. This pointer will be the return value if the call succeeds.

destination_length
(Input) The length of the destination buffer.

Authorities

Authorization of *R (allow access to the object) to the host aliases file specified by the *HOSTALIASES* environment variable.

You also need *X authority to each directory in the path of the host aliases file.

Return Value

(NULL) No alias found or an error occurred.

(destination) A pointer to the destination buffer updated with the alias found.

Error Conditions

When the *res_hostalias()* function fails, *errno* can be set to one of the following:

[EACCES]

Permission denied. The process does not have the appropriate privileges to the host aliases file specified by the *HOSTALIASES* environment variable.

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

One of the following reasons:

- The **res** appears to have been previously initialized but the reserved field is not set to zeros or an input pointer was NULL.
- An alias was found that contains a period.

Usage Notes

1. If the *RES_NOALIASES* option is set, no processing is done and a NULL will be returned.
2. If the res structure has not been initialized, *res_ninit()* will be called.

Related Information

- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

res_init()—Initialize _res Structure

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_init(void)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_init()* function is used to initialize the **_res** structure for name resolution. Two bits are set in the structure to indicate that it has been initialized. (These are the RES_INIT and RES_XINIT bits in the options field of the **_res** structure.) Also, the default domain name and other components of the domain to search are put into the **_res** structure.

The **_res** structure is defined in **<resolv.h>**.

```
struct state {
  int    retrans;
  int    retry;
  long   options;
  int    nscount;
  struct sockaddr_in nsaddr_list[MAXNS];
  u_short id;
  char   defdname[MAXDNAME];
  char   reserved0[1];
  char   reserved1[13];
  char   *dnsrch[MAXDNSRCH+1];

  /* Extended state structure begins here.*/
  struct {
              struct in_addr addr;
              uint           mask;
  }        sort_list[MAXRESOLVSORT];
  int      res_h_errno;
  int      extended_error;
  unsigned ndots:4;
  unsigned nsort:4;
  char     state_data[27];
  int      internal_use[4];
  char     reserved[444];
  };


#define nsaddr nsaddr_list[0]

extern struct state _res;
```

retrans Time interval in seconds between retries. The default is received from QUSRSYS/QATOCTCPIP which is configured with the Change TCP/IP Domain (CHGTCPDMN) command

retry Number of times to retransmit. The default is received from QUSRSYS/QATOCTCPIP which is configured with the Change TCP/IP Domain (CHGTCPDMN) command

options Contains flag bits to indicate the different resolver options. The default is *RES_DEFAULT*

nscount
Number of name servers. *res_ninit()* sets the number of name servers to the number found in the database file. The maximum is 3

nsaddr_list
Contains the address(es) of the name server(s)

id Current packet ID. The id is initialized to a random number

defdname
Default domain name or the search list

dnsrch Contains the components of the search list. By default it points to components of *defdname* which contains the local domain or the configured search list. However a program may allocate separate storage for a customized search list and set the elements of *dnsrch* to point to it. Each component pointed to by an element of *dnsrch* must be NULL terminated.

sort_list
List of address/mask pairs that will be used to sort the results of a *gethostbyname()* or *gethostbyname_r()* operation

res_h_errno
Holds the last *h_errno* or *errno* set by the resolver for this context

ndots Number of dots in a name that will trigger an absolute query instead of using the *dnsrch*

nsort Number of elements in the *sort_list* array

state_data
Used internally by the resolver

reserved0,*reserved1* **and** *reserved*
Fields are that set to zeros by *res_ninit()* or *res_init()*. If the **res** structure is manually initialized by a program, it also must set these structures to zeros.

nsaddr Defined for backward compatibility

options The value for the *options* is constructed by performing an OR operation on the following values:

RES_INIT	Indicates that the **res** structure has been initialized.
RES_AAONLY	Requests the answer be authoritative and not from a name server's cache.
RES_USEVC	Tells the resolver to use TCP instead of UDP.
RES_IGNTC	Tells the resolver to ignore truncation.
RES_RECURSE	Specifies that recursion is desired.
RES_DEFNAMES	Appends the default domain name to single label queries.
RES_STAYOPEN	Causes the TCP connection to remain open (used with RES_USEVC).
RES_DNSRCH	Searches using *dnsrch*.
RES_INSECURE1	Disables type 1 security. Type 1 security rejects responses that didn't come from one of the configured DNS servers.
RES_INSECURE2	Disables type 2 security. Type 2 security checks the question section of the reply to ensure it matches the original query sent.
RES_NOALIASES	Tells the resolver to ignore the HOSTALIASES environment variable.
RES_ROTATE	Tells the resolver to rotate through the list of DNS servers (*nsaddr_list*).
RES_NOCHECKNAME	Tells the resolver not to check host names in replies for disallowed characters such as underscore (_), non-ASCII, or control characters.
RES_KEEPTSIG	Stops the resolver from stripping TSIG records on replies.
RES_NOCACHE	Do not look in the resolver answer cache. Query the name server. The answer may still be locally cached.

The following four values are i5/OS specific.

RES_XINIT	Indicates that the extended portion of the **res** structure has been initialized.
RES_CP850	Use ASCII code page 850 and not ASCII code page 819.
RES_RETRYTCP	Retry with a TCP connection if the UDP connection fails for any reason.
RES_NSADDRONLY	Only use the list of addresses in *nsaddr*. There may be a separate SOCKS DNS configured that would normally be used.
RES_DEFAULT	This is the default. Causes an OR operation on the RES_RECURSE, RES_DEFNAMES, RES_DNSRCH values.

Authorities:

No authorization is required.

Return Value

None.

Error Conditions

res_init() can set *errno* to the following:

[EINVAL]

_res appears to have been previously initialized but the reserved field is not set to zeros.

[EUNKNOWN]

res_init() was unable to retrieve the DNS server configuration.

Usage Notes

1. If no entry was configured with Change TCP/IP Domain (CHGTCPDMN), then *res_init()* does the following:
   - Calls *gethostname()* to get the default domain name. The default domain name in this case is the host name minus the first component of the name. For example, if the host name is ABC.RCHLAND.IBM.COM, the default name is RCHLAND.IBM.COM.
   - Calls *getservbyname()* to get the port number.
   - Uses hard-coded defaults for *retrans*, *retry* and *ndots* (5, 4 and 1 respectively).
2. The default initialization values can be overridden with enviroment variables. *Note:*The name of the environment variable must be uppercased. The string value may be mixed case. Japanese systems using CCSID 290 should use uppercase characters and numbers only in both environment variables names and values.
   - LOCALDOMAIN

     The configured search list (struct state.defdname and struct state.dnsrch) can be overridden by setting the environment variable LOCALDOMAIN to a space-separated list of up to 6 search domains with a total of 256 characters (including spaces). If a search list is specified, the default local domain is not used on queries.
   - RES_OPTIONS allows certain internal resolver variables to be modified. The environment variable can be set to one or more of the following space-separated options:
     – NDOTS:n sets a threshold for the number of dots which must appear in a name given to res_query() before an initial absolute query will be made. The default for n is ``1'', meaning that if there are any dots in a name, the name will be tried first as an absolute name before any search list elements are appended to it.
     – TIMEOUT:n sets the amount of time (in seconds) the resolver will wait for a response from a remote name server before giving up and retrying the query.
     – ATTEMPTS:n sets the number of queries the resolver will send to a given nameserver before giving up and trying the next listed nameserver.
     – ROTATE sets RES_ROTATE in _res.options , which causes round robin selection of nameservers from among those listed. This has the effect of spreading the query load among all listed servers, rather than having all clients try the first listed server first every time.
     – NO-CHECK-NAMES sets RES_NOCHECKNAME in _res.options , which disables the modern BIND checking of incoming host names and mail names for invalid characters such as underscore (_), non-ASCII, or control characters.
   - QIBM_BIND_RESOLVER_FLAGS

     The RES_DEFAULT options (struct state.options) and system configured values (Change TCP/IP Domain - CHGTCPDMN) can be overridden by setting the environment variable QIBM_BIND_RESOLVER_FLAGS to a space separated list of resolver option flags. The state.options structure will be initialized normally, using RES_DEFAULT, OPTIONS environment values and CHGTCPDMN configured values. Then this environment varible will be used to override those defaults. The flags named in this environment variable may be prepended with a '+', '-' or 'NOT_'

to set ('+') or reset ('-','NOT_') the value. For example, to turn on RES_NOCHECKNAME and turn off RES_ROTATE:
ADDENVVAR ENVVAR(QIBM_BIND_RESOLVER_FLAGS) VALUE('RES_NOCHECKNAME NOT_RES_ROTATE')
or
ADDENVVAR ENVVAR(QIBM_BIND_RESOLVER_FLAGS) VALUE('+RES_NOCHECKNAME -RES_ROTATE')

- QIBM_BIND_RESOLVER_SORTLIST

  A sort list (struct state.sort_list) can be configured by setting the environment variable QIBM_BIND_RESOLVER_SORTLIST to a space-separated list of up to 10 ip addresses/mask pairs in dotted decimal format (9.5.9.0/255.255.255.0)

Note: Environment variables are only checked after a successful call to *res_init()* or *res_ninit()*. So if the structure has been manually initialized, environment variables are ignored. Also note that the structure is only initialized once so later changes to the environment variables will be ignored.

3. *res_init()* is called by *res_send()*, *res_mkquery()*, *res_search()*, and *res_query()* if they detect the **_res** structure has not been initialized (RES_INIT option). *res_init()* can also be called directly to change the defaults and hence, change the behavior of one of the above routines. For example, if you want to use TCP rather than attempt UDP first, simply call *res_init()* directly. Then before the call to *res_send()*, set the RES_USEVC bit in the options flag. Other things in the **_res** structure, like the number of retries or time interval between retries, can be changed in a like manner.
4. If the server protocol configured with Change TCP/IP Domain (CHGTCPDMN) is set to TCP, then *res_init()* sets the RES_USEVC bit in the options field of the **_res** structure.
5. In a thread-enabled environment the **_res** structure is shared among all threads within a process.

Related Information

- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_close()—Close Socket and Reset _res Structure" on page 287—Close Socket and Reset _res Structure
- "res_mkquery()—Place Domain Query in Buffer"—Place Domain Query in Buffer
- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_search()—Search for Domain Name" on page 318—Search for Domain Name
- "res_send()—Send Buffered Domain Query or Update" on page 320—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

res_mkquery()—Place Domain Query in Buffer

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_mkquery(int  operation,
                char *domain_name,
                int  class,
                int  type,
                char *search_data,
```

```
                int  search_data_length,
                struct rrec *reserved,
                char *query_buffer,
                int  query_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_mkquery()* function is used to make standard query messages (DNS packets) for name servers.

Parameters

operation
(Input) The query operation desired. This gets put into OPCODE in the header of the packet. Common values are listed below (see <arpa/nameser.h> for all possible values):

ns_o_query or *QUERY*	Standard query request. (This value is almost always used.)

domain_name
(Input) The pointer to the name of the domain.

class (Input) The class of data being looked for. Common values are listed below (see <arpa/nameser.h> for all possible values):

ns_c_in or *C_IN*	Specifies the ARPA Internet.
ns_c_any or *C_ANY*	This is the wildcard match.

type (Input) The type of request being made. Common values are listed below (see <arpa/nameser.h> for all possible values):

ns_t_a or *T_A*	Host address.
ns_t_aaaa	IPv6 address.
ns_t_ns or *T_NS*	Authoritative server.
ns_t_cname or *T_CNAME*	Canonical name.
ns_t_soa or *T_SOA*	Start of authority zone.
ns_t_wks or *T_WKS*	Well-known service.
ns_t_ptr or *T_PTR*	Domain name pointer.
ns_t_hinfo or *T_HINFO*	Host information.
ns_t_mx or *T_MX*	Mail routing information.
ns_t_txt or *T_TXT*	Text strings.
ns_t_any or *T_ANY*	Wildcard match.

search_data
(Input) A buffer containing the data for inverse queries. It is NULL for types other than IQUERY.

search_data_length
(Input) The length of *search_data*. It is NULL for types other than IQUERY.

reserved
(Input) A reserved and currently unused parameter. It is always a NULL pointer (defined for compatibility).

query_buffer
(Output) A pointer to a user-supplied location containing the query message.

query_buffer_length
(Input) The length of *query_buffer*.

Authorities:

No authorization is required.

Return Value

res_mkquery() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the size of the query.

Error Conditions

When the *res_mkquery()* function fails, *errno* can be set to one of the following:

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

The **_res** appears to be initialized but the reserved field is not set to zeros.

[EMSGSIZE]

The message buffer was too small. The query was larger than the value of *query_buffer_length*

Usage Notes

1. *res_mkquery()* creates a standard query message (DNS packet). It fills in the header fields, compresses the domain name into the question section, and fills in the other question fields. This query message is placed in *query_buffer*.
2. *res_mkquery()* calls *res_init()* if the **_res** structure has not been initialized.
3. *res_mkquery()* expects EBCDIC data as input. The output from *res_mkquery()* is also EBCDIC.
4. In a thread-enabled environment, the **_res** structure is shared among all threads within a process.

Related Information

- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_close()—Close Socket and Reset _res Structure" on page 287—Close Socket and Reset _res Structure
- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_search()—Search for Domain Name" on page 318—Search for Domain Name
- "res_send()—Send Buffered Domain Query or Update" on page 320—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

res_nclose()—Close Socket and Reset res Structure

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_nclose(state* res)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nclose()* function is similar to *res_close()* but it uses a user-declared **res** pointer instead of the shared **_res**.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "res_close()—Close Socket and Reset _res Structure" on page 287.

Parameters

res (Input) The pointer to the **state** structure.

Related Information

- "res_close()—Close Socket and Reset _res Structure" on page 287—Close Socket and Reset _res Structure
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure"—Initialize res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V5R1

res_ninit()—Initialize res Structure

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_ninit(state* res)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_ninit()* function is similar to *res_init()* but it uses a user-declared **res** pointer instead of the shared **_res**.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure.

Parameters

res (Input/Output) The pointer to the **state** structure.

The RES_INIT and RES_XINIT options flags must be initialized to zero before the first call to any resolver API or the **res** structure will not be properly initialized. For example:

```
state res;
res.options &= ~ (RES_INIT | RES_XINIT);
int n = res_ninit(&res);
```

Return Value

res_ninit() returns an integer. Possible values are:

- -1 (unsuccessful)
- 0 (successful)

Error Conditions

When the *res_ninit()* function fails, *errno* can be set to one of the following:

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

The **res** appears to have been previously initialized but the reserved field is not set to zeros.

Related Information

- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query

- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how **res_ninit()** is used and how initialization defaults can be changed after initialization:

```
#include <stdio.h>
#include <errno.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
#include <netdb.h>

/* Declare update records - a zone record, a pre-requisite record, and
   an update record */
 ns_updrec update_records[] =
 {
    {
       {NULL,NULL},
       {NULL,&update_records[1]},
       ns_s_zn,                  /* a zone record */
       "mydomain.ibm.com.",
       ns_c_in,
       ns_t_soa,
       0,
       NULL,
       0,
       0,
       NULL,
       NULL,
       0
    },
    {
       {NULL,NULL},
       {&update_records[0],&update_records[2]},
       ns_s_pr,                  /* pre-req record */
       "mypc.mydomain.ibm.com.",
       ns_c_in,
       ns_t_a,
       0,
       NULL,
       0,
       ns_r_nxdomain,            /* record must not exist */
       NULL,
       NULL,
       0
    },
    {
       {NULL,NULL},
       {&update_records[1],NULL},
       ns_s_ud,                 /* update record */
       "mypc.mydomain.ibm.com.",
       ns_c_in,
       ns_t_a,
       10,
       (unsigned char *)"10.10.10.10",
       11,
       ns_uop_add,              /* to be added  */
       NULL,
       NULL,
       0
```

```
    }
 };

void main()
{
   struct state res;
   int result;
   unsigned char update_buffer[2048];
   int buffer_length = sizeof update_buffer;

   unsigned char answer_buffer[2048];

   /* Turn off the init flags so that the structure will be initialized
   */
   res.options &= ~ (RES_INIT | RES_XINIT);

   result = res_ninit(&res);

   /* Put processing here to check the result and handle errors
   */

   /* We choose to use TCP and not UDP, so set the appropriate option now
      that the res variable has been initialized.
   */
   res.options |= RES_USEVC;

   /* Send a query for mypc.mydomain.ibm.com address records
   */
   result = res_nquerydomain(&res,"mypc", "mydomain.ibm.com.", ns_c_in, ns_t_a,
                             update_buffer, buffer_length);

   /* Sample error handling and printing errors
   */
   if (result == -1)
   {
      printf("\nquery domain failed.  result = %d \nerrno: %d: %s \nh_errno: %d: %s",
             result,
             errno, strerror(errno),
             h_errno, hstrerror(h_errno));
      return;
   }

   /* The output on a failure will be:

      query domain failed.  result = -1
      errno: 0: There is no error.
      h_errno: 5: Unknown host

   */

   {
      /* Build an update buffer (packet to be sent) from the update records
      */
      result = res_nmkupdate(&res, update_records, update_buffer, buffer_length);

      /* Put processing here to check the result and handle errors
      */
   }

   {
   char zone_name[NS_MAXDNAME];
   size_t zone_name_size = sizeof zone_name;
   struct sockaddr_in s_address;
   struct in_addr addresses[1];
   int number_addresses = 1;

   /* Find the DNS server that is authoritative for the domain
```

```
      that we want to update
   */

   result = res_findzonecut(&res, "mypc.mydomain.ibm.com", ns_c_in, 0,
                            zone_name, zone_name_size,
                            addresses, number_addresses);

   /* Put processing here to check the result and handle errors
   */

   /* Check if the DNS server found is one of our regular
      DNS addresses
   */

   s_address.sin_addr = addresses[0];
   s_address.sin_family = res.nsaddr_list[0].sin_family;
   s_address.sin_port = res.nsaddr_list[0].sin_port;
   memset(s_address.sin_zero, 0x00, 8);

   result = res_nisourserver(&res, &s_address);

   /* Put processing here to check the result and handle errors
   */

   /* Set the DNS address found with res_findzonecut into the res
      structure.  We will send the (TSIG signed) update to that DNS.
   */

   res.nscount = 1;
   res.nsaddr_list[0] = s_address;
   }

   {
      ns_tsig_key my_key = {
         "my-long-key",            /* This key must exist on the DNS */
         NS_TSIG_ALG_HMAC_MD5,
         (unsigned char*) "abcdefghijklmnopqrstuvwx",
         24
      };

      /* Send a TSIG signed update to the DNS
      */
      result = res_nsendsigned(&res, update_buffer, result,
                               &my_key,
                               answer_buffer, sizeof answer_buffer);

      /* Put processing here to check the result and handle errors
      */
   }

   /* The res_findzonecut(), res_nmkupdate(), and res_nsendsigned() could
      be replaced with one call to res_nupdate() using update_records[1]
      to skip the zone record::

      result = res_nupdate(&res, &update_records[1], &my_key);

   */
   return;
}
```

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

res_nisourserver()—Check Server Address

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nisourserver(state* res,
                     const struct sockaddr_in server)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nisourserver()* looks up the specified *server* address in the ns_addr_list[] of the specified *res* structure.

Parameters

res (Input) The pointer to the **state** structure.

server (Input) The server address to check.

Authorities:

No authorization is required.

Return Value

(0) Server not found in ns_addr_list[].

(>0) Server found in ns_addr_list[].

(<0) Error.

Error Conditions

When the *res_nisourserver()* function returns an error, *errno* will be set to one of the following:

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

One of the following reasons:

- A NULL pointer was passed to *res_nisourserver()*
- The **res** appears to be initialized but the reserved field is not set to zeros.

Related Information

- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name

- “res_nsend()—Send Buffered Domain Query or Update” on page 310—Send Buffered Domain Query
- “res_xlate()—Translate DNS Packets” on page 323—Translate DNS Packets

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

res_nmkquery()—Place Domain Query in Buffer

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nmkquery(state* res,
                 int  operation,
                 const char *domain_name,
                 int  class,
                 int  type,
                 const unsigned char *search_data,
                 int  search_data_length,
                 const unsigned char *reserved,
                 unsigned char *query_buffer,
                 int  query_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nmkquery()* function is similar to *res_mkquery()* but it uses a user-declared **res** pointer instead of the shared **_res**.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see “res_mkquery()—Place Domain Query in Buffer” on page 296—Place Domain Query in Buffer.

Parameters

res (Input/Output) The pointer to the **state** structure.

Related Information

- “res_mkquery()—Place Domain Query in Buffer” on page 296—Place Domain Query in Buffer
- “res_findzonecut()—Find the Enclosing Zone and Servers” on page 288—Find the Enclosing Zone and Servers
- “res_hostalias()—Retrieve the host alias” on page 291—Retrieve the host alias
- “res_ninit()—Initialize res Structure” on page 299—Initialize res Structure
- “res_nclose()—Close Socket and Reset res Structure” on page 299—Close Socket and Reset res Structure
- “res_nquery()—Send Domain Query” on page 307—Send Domain Query
- “res_nsearch()—Search for Domain Name” on page 309—Search for Domain Name
- “res_nsend()—Send Buffered Domain Query or Update” on page 310—Send Buffered Domain Query
- “res_xlate()—Translate DNS Packets” on page 323-Translate DNS Packets

API introduced: V3R1

res_nmkupdate()—Construct an Update Packet

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nmkupdate(state* res,
             ns_updrec *update_record,
             unsigned char *buffer,
             int buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nmkupdate()* function builds a dynamic update packet from the linked list of update records.

Parameters

res (Input) The pointer to the **state** structure.

update_record
(Input) The pointer to the linked list of update records. See "res_nupdate()—Build and Send Dynamic Updates" on page 314 for more information.

buffer (Input) The pointer to the buffer to be filled in with the update packet.

buffer_length
(Input) The length of the ***buffer***.

Authorities

No authorization is required.

Return Value

res_nmkupdate() returns an integer. Possible values are:

- n (successful), where n is the actual size of the resulting update packet.
- -1 (unsuccessful) An error occurred parsing a word or number in the rdata portion of the update records.
- -2 (unsuccessful) The buffer was too small
- -3 (unsuccessful) The zone section is not the first section in the linked list, or the section order has a problem. The section order is *ns_s_zn*, *ns_s_pr* and *ns_s_ud*.
- -4 (unsuccessful) A number overflow occurred.
- -5 (unsuccessful) Unknown operation or no records found.

Error Conditions

When the *res_nmkupdate()* function fails, *res_nmkupdate()* can set *errno* to one of the following:

[ECONVERT]

Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

One of the following reasons:

- An invalid length or NULL pointer was passed to *res_nmkupdate()*
- The **res** appears to be initialized but the reserved field is not set to zeros.

Note: No attempt is made to initialize the **res** structure if it was initialized previous to the *res_nmkupdate()* being issued.

[EMSGSIZE]

The message buffer was too small. The return value was -2.

Usage Notes

1. *res_nmkupdate()* calls *res_ninit()* if the **res** structure has not been initialized.
2. *res_nmkupdate()* assumes that the data passed to it is EBCDIC and is in the default coded character set identifier (CCSID) currently in effect for the job. It translates the data from the default CCSID currently in effect for the job to ASCII (CCSID 819) before the data is sent out to a name server. The response that it receives from the name server is returned in the default CCSID currently in effect for the job.

Related Information

- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nquery()—Send Domain Query"—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_nsendsigned()—Send Authenticated Domain Query or Update" on page 311—Send Authenticated Domain Query
- "res_nupdate()—Build and Send Dynamic Updates" on page 314—Build and Send Dynamic Updates
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

res_nquery()—Send Domain Query

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

```
int res_nquery(state* res,
               const char *domain_name,
               int  class,
               int  type,
               unsigned char *answer_buffer,
               int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nquery()* function is similar to *res_query()* but it uses a user-declared **res** pointer instead of the shared **_res**.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "res_query()—Send Domain Query" on page 316—Send Domain Query.

Parameters

res (Input/Output) The pointer to the **state** structure.

Related Information

- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

res_nquerydomain()—Send 2 String Domain Query

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_nquerydomain(state* res,
                      const char *string1,
                      const char *string2,
                      int  class,
                      int  type,
                      unsigned char *answer_buffer,
                      int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nquerydomain()* concatenates *string1 + string2* into a new *domain_name* parameter and calls *res_nquery()*. For more information on *domain_name*, the remaining parameters, authorities required, return values, and related information, see "res_nquery()—Send Domain Query" on page 307.

Parameters

string1
(Input) The pointer to the first string. In practice this is generally a host name.

string2
(Input) The pointer to the first string. In practice this is generally a zone name.

Related Information

- "res_nquery()—Send Domain Query" on page 307—Send Domain Query

Example

See Code disclaimer information for information pertaining to code examples.

See "res_ninit()—Initialize res Structure" on page 299 for an example of how *hstrerror()* is used.

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

res_nsearch()—Search for Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nsearch(state* res,
                const char *domain_name,
                int  class,
                int  type,
                unsigned char *answer_buffer,
                int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nsearch()* function is similar to *res_search()* but it uses a user-declared **res** pointer instead of the shared **_res**.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see "res_search()—Search for Domain Name" on page 318—Search for Domain Name.

Parameters

res (Input/Output) The pointer to the **state** structure.

Related Information

- “res_search()—Search for Domain Name” on page 318—Search for Domain Name
- “res_findzonecut()—Find the Enclosing Zone and Servers” on page 288—Find the Enclosing Zone and Servers
- “res_hostalias()—Retrieve the host alias” on page 291—Retrieve the host alias
- “res_ninit()—Initialize res Structure” on page 299—Initialize res Structure
- “res_nmkquery()—Place Domain Query in Buffer” on page 305—Place Domain Query in Buffer
- “res_nquery()—Send Domain Query” on page 307—Send Domain Query
- “res_nclose()—Close Socket and Reset res Structure” on page 299—Close Socket and Reset res Structure
- “res_nsend()—Send Buffered Domain Query or Update”—Send Buffered Domain Query
- “res_xlate()—Translate DNS Packets” on page 323—Translate DNS Packets

API introduced: V3R1

---

res_nsend()—Send Buffered Domain Query or Update

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nsend(state* res,
              const unsigned char *query_buffer,
              int  query_buffer_length,
              unsigned char *answer_buffer,
              int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nsend()* function is similar to *res_send()* but it uses a user-declared **res** pointer instead of the shared **_res**.

For a description of this function and more information on the parameters, authorities required, return values, error conditions, error messages, usage notes, and related information, see “res_send()—Send Buffered Domain Query or Update” on page 320—Send Buffered Domain Query.

Parameters

res (Input/Output) The pointer to the **state** structure.

Related Information

- “res_send()—Send Buffered Domain Query or Update” on page 320—Send Buffered Domain Query
- “res_findzonecut()—Find the Enclosing Zone and Servers” on page 288—Find the Enclosing Zone and Servers
- “res_hostalias()—Retrieve the host alias” on page 291—Retrieve the host alias
- “res_ninit()—Initialize res Structure” on page 299—Initialize res Structure
- “res_nmkquery()—Place Domain Query in Buffer” on page 305—Place Domain Query in Buffer

- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

---

res_nsendsigned()—Send Authenticated Domain Query or Update

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nsendsigned(state* res,
             const unsigned char *query_buffer,
             int  query_buffer_length,
             ns_tsig_key * key,
             unsigned char *answer_buffer,
             int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nsendsigned()* function is similar to *res_nsend()* but it uses the specified key to create a transaction signature (TSIG) to sign the query or update packet and to authenticate the response.

Parameters

res (Input) The pointer to the **state** structure.

query_buffer
(Input) The pointer to the query or update message.

query_buffer_length
(Input) The length of *query_buffer*.

key (Input) The pointer to the key to use for authentication. This key must exist on the name server.

answer_buffer
(Output) The pointer to where the response is stored.

answer_buffer_length
(Input) The size of the answer_buffer.

Authorities

No authorization is required.

Return Value

res_nsendsigned() returns an integer. Possible values are:

- n (successful), where n is the actual size of the answer returned.
- -1 (unsuccessful)
- -ns_r_badkey (unsuccessful) The key was invalid or the signing failed.

- NS_TSIG_ERROR_NO_SPACE (unsuccessful) The message buffer was too small to add the TSIG.

Error Conditions

When the *res_nsendsigned()* function fails, *res_nsendsigned()* can set *errno* to one of the following:

[ECONNREFUSED]

Not able to connect to a server.

[ECONVERT]

Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

One of the following reasons:

- An invalid length or NULL pointer was passed to *res_nsendsigned()*
- The **res** appears to be initialized but the reserved field is not set to zeros.

Note: No attempt is made to initialize the **res** structure if it was initialized previous to the *res_nsendsigned()* being issued.

[EMSGSIZE]

The message buffer was too small to add the TSIG. The return value was NS_TSIG_ERROR_NO_SPACE.

[ENOTTY]

The message or reply couldn't be verified. See *extended_error* in the **res** structure:

NS_TSIG_ERROR_FORMERR

The message is malformed.

NS_TSIG_ERROR_NO_TSIG

The message does not contain a TSIG record.

NS_TSIG_ERROR_ID_MISMATCH

The TSIG original ID field does not match the message ID.

(-ns_r_badkey)

Verification failed due to an invalid key.

(-ns_r_badsig)

Verification failed due to an invalid signature.

(-ns_r_badtime)

Verification failed due to an invalid timestamp.

ns_r_badkey

Verification succeeded but the message had an error (rcode) of *ns_r_badkey*.

ns_r_badsig

Verification succeeded but the message had an error (rcode) of *ns_r_badsig*.

ns_r_badtime

Verification succeeded but the message had an error (rcode) of *ns_r_badtime*.

[ETIMEDOUT]

A timeout received from a connected server.

When the *res_nsearch()* function fails, *h_errno* (defined in **<netdb.h>**) can also be set to one of the following:

HOST_NOT_FOUND

Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

NO_RECOVERY

An invalid length or NULL pointer was passed to *res_nsendsigned()* or the **res** could not be initialized properly.

Notes:

- No attempt is made to initialize the **res** structure if it was initialized previous to the *res_nsendsigned()* being issued.
- There are numerous other values that *errno* can be set to by the sockets functions that *res_nsendsigned()* calls. The above values are the only values that *res_nsendsigned()* can specifically set. Refer to other sockets functions for the other values. *errno* is always set in an error condition, but *h_errno* is not necessarily set.

  After receiving an error reply packet, *res_nsendsigned()* will set the *extended_error* field in the *state* structure to the last reply return code from the DNS server. See <arpa/nameser.h> for all possible values of *ns_rcode*.

Usage Notes

1. *res_nsendsigned()* sends the query or update to the local name server and handles all timeouts and retries. The response packet is stored in *answer_buffer*.
2. *res_nsendsigned()* calls *res_ninit()* if the **res** structure has not been initialized.
3. *res_nsendsigned()* uses the UDP protocol, except for the following cases in which it uses TCP to send the packet.
   - If the RES_USEVC or RES_STAYOPEN bits are set in the options field of the **res** structure.
   - If the configuration from Change TCP/IP Domain (CHGTCPDMN) specifies that the server protocol is TCP.
   - If the truncation bit is set in the packet header on the response from a UDP packet, and RES_IGNTC is not set in the **res** structure.
4. *res_nsendsigned()* does not perform iterative queries and expects the name server to handle recursion.
5. *res_nsendsigned()* assumes that the data passed to it is EBCDIC and is in the default coded character set identifier (CCSID) currently in effect for the job. It translates the data from the default CCSID currently in effect for the job to ASCII (CCSID 819) before the data is sent out to a name server. The response that it receives from the name server is returned in the default CCSID currently in effect for the job.
6. *res_nsendsigned()* will not use the local cache. It will always send the packet to the server.
7. When using TSIG, it is important that the QUTCOFFSET system value is set correctly for the local time zone. The resolver system and name server timestamps must be within 5 minutes of each other (adjusted by the UTC offset) or the authentication will fail with *ns_r_badtime*.

Related Information

- “hstrerror()—Retrieve Resolver Error Message” on page 267—Retrieve Resolver Error Message
- “res_nclose()—Close Socket and Reset res Structure” on page 299—Close Socket and Reset res Structure
- “res_findzonecut()—Find the Enclosing Zone and Servers” on page 288—Find the Enclosing Zone and Servers
- “res_hostalias()—Retrieve the host alias” on page 291—Retrieve the host alias
- “res_ninit()—Initialize res Structure” on page 299—Initialize res Structure
- “res_nmkquery()—Place Domain Query in Buffer” on page 305—Place Domain Query in Buffer
- “res_nmkupdate()—Construct an Update Packet” on page 306—Construct an Update Packet
- “res_nquery()—Send Domain Query” on page 307—Send Domain Query
- “res_nsearch()—Search for Domain Name” on page 309—Search for Domain Name
- “res_nsend()—Send Buffered Domain Query or Update” on page 310—Send Buffered Domain Query
- “res_nupdate()—Build and Send Dynamic Updates”—Build and Send Dynamic Updates
- “res_xlate()—Translate DNS Packets” on page 323—Translate DNS Packets

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

res_nupdate()—Build and Send Dynamic Updates

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_nupdate(state* res,
             ns_updrec *update_record
             ns_tsig_key *key)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_nupdate()* function separates the linked list of update records into groups so that all records in a group will belong to a single zone on the nameserver. It creates a dynamic update packet for each zone and sends it to the nameservers for that zone.

Parameters

res (Input) The pointer to the **state** structure.

update_record
(Input) The pointer to the linked list of update records.

key (Input) The pointer to the key to use for authentication. If it is NULL, no authentication will be done.

The **ns_updrec** structure is defined in **<arpa/nameser.h>**.

```
struct ns_updrec {
   struct {
      struct ns_updrec *prev, *next;
   } r_link, r_glink;
   ns_sect               r_section;
```

```
   char *                r_dname;
   ns_class              r_class;
   ns_type               r_type;
   uint32                r_ttl;
   unsigned char *       r_data;
   uint32                r_size;
   int32                 r_opcode;
/* The following fields are ignored by the resolver routines  */
   struct databuf *      r_dp;
   struct databuf *      r_deldp;
   uint32                r_zone;
};

typedef struct ns_updrec ns_updrec;
```

r_link **and** *r_glink*
: Doubly linked lists of **ns_updrec** records. *res_nupdate()* uses *r_link* as its list of records to process and ignores *r_glink*. *res_nmkupdate()* uses *r_glink* as its list of records to process and ignores *r_link*.

r_section
: See the *ns_sect* enums in **<arpa/nameser.h>** for allowed values.

r_dname,*r_class*,*r_type*, *r_ttl*,*r_data*, **and** *r_size*
: Identify the resource record to the DNS

r_opcode
: Type of update operation. Valid operations are *ns_uop_delete* or *ns_uop_add*

These fields are ignored by the resolver: *r_dp*, *r_deldp*, *r_zone*.

Authorities

No authorization is required.

Return Value

res_nupdate() returns an integer. Possible values are:

- n (successful), where n is the number of zones updated.
- -1 (unsuccessful)

Error Conditions

When the *res_nupdate()* function fails, *res_nupdate()* can set *errno* to one of the following:

[ECONVERT]

: Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

[EFAULT]

: The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

: One of the following reasons:
  - An invalid length or NULL pointer was passed to *res_nupdate()*
  - The **res** appears to be initialized but the reserved field is not set to zeros.

Notes:

- No attempt is made to initialize the **res** structure if it was initialized previous to the *res_nupdate()* being issued.
- *res_nupdate()* calls *res_findzonecut()*, *res_nmkupdate()* and *res_nsend()* or *res_nsendsigned()* so errnos from those routines may also be set.

Usage Notes

1. *res_nupdate()* calls *res_ninit()* if the **res** structure has not been initialized.
2. *res_nupdate()* calls *res_findzonecut()* to find the zone and name server to be updated for each input record and sorts the records by zone. Then it makes a zone record for each zone and prepends it to the update records. It calls *res_nmkupdate()* to make the update packet and then calls either *res_nsend()* or *res_nsendsigned()* to send the packet. Note that since *res_nupdate()* prepends a new zone record, the input records must only contain pre-requisite and update records, not zone records.
3. *res_nupdate()* assumes that the data passed to it is EBCDIC and is in the default coded character set identifier (CCSID) currently in effect for the job. It translates the data from the default CCSID currently in effect for the job to ASCII (CCSID 819) before the data is sent out to a name server. The response that it receives from the name server is returned in the default CCSID currently in effect for the job.
4. *res_nupdate()* will not use the local cache. It will always send the packet to the server.
5. When using TSIG, it is important that the QUTCOFFSET system value is set correctly for the local time zone. The resolver system and name server timestamps must be within 5 minutes of each other (adjusted by the UTC offset) or the authentication will fail with *ns_r_badtime*.

Related Information

- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nmkupdate()—Construct an Update Packet" on page 306—Construct an Update Packet
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_nsendsigned()—Send Authenticated Domain Query or Update" on page 311—Send Authenticated Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V5R1

Top | UNIX-Type APIs | APIs by category

---

res_query()—Send Domain Query

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query(char *domain_name,
              int  class,
              int  type,
              char *answer_buffer,
              int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_query()* function is used to interface to the server query mechanism.

Parameters

domain_name
(Input) The pointer to the domain name.

class (Input) The class of data being looked for. See "res_mkquery()—Place Domain Query in Buffer" on page 296 or <arpa/nameser.h> for possible values.

type (Input) The type of request being made. See "res_mkquery()—Place Domain Query in Buffer" on page 296 or <arpa/nameser.h> for possible values.

answer_buffer
(Output) The pointer to an address where the response is stored.

answer_buffer_length
(Input) The size of the answer area.

Authorities

No authorization is required.

Return Value

res_query() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the actual size of the answer returned.

Error Conditions

When the *res_query()* function fails, *errno* can be set to one of the following:

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

The **_res** appears to be initialized but the reserved field is not set to zeros.

When the *res_query()* function fails, *h_errno* (defined in **<netdb.h>**) can be set to one of the following:

[HOST_NOT_FOUND]

The domain name specified by the *domain_name* parameter was not found. The return code in the response packet was NXDOMAIN.

[TRY_AGAIN]

Either the name server is not running or the name server returned SERVFAIL in the response packet.

[NO_RECOVERY]

An unrecoverable error has occurred. Either the domain name could not be compressed because it was invalid or the name server returned FORMERR, NOTIMP, or REFUSED.

[NO_DATA]

The domain name exists but there is no data of the requested type.

Usage Notes

1. *res_query()* makes a query packet by calling *res_mkquery()*, sends the query by calling *res_send()*, and makes preliminary checks on the reply. The reply message is left in *answer_buffer*.
2. *res_query()* calls *res_init()* if the **_res** structure has not been initialized.
3. *res_query()* expects EBCDIC data as input. The output from *res_query()* is also EBCDIC.
4. In a thread-enabled environment, the **_res** structure is shared among all threads within a process.

Related Information

- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_mkquery()—Place Domain Query in Buffer" on page 296—Place Domain Query in Buffer
- "res_close()—Close Socket and Reset _res Structure" on page 287—Close Socket and Reset _res Structure
- "res_search()—Search for Domain Name"—Search for Domain Name
- "res_send()—Send Buffered Domain Query or Update" on page 320—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

res_search()—Search for Domain Name

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_search(char *domain_name,
               int  class,
               int  type,
               char *answer_buffer,
               int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_search()* function is used to make a query message and wait for a response.

Parameters

domain_name
(Input) The pointer to the domain name.

class (Input) The class of data being looked for. See “res_mkquery()—Place Domain Query in Buffer” on page 296 or <arpa/nameser.h> for possible values.

type (Input) The type of request being made. See “res_mkquery()—Place Domain Query in Buffer” on page 296 or <arpa/nameser.h> for possible values.

answer_buffer
(Output) The pointer to an address where the response is stored.

answer_buffer_length
(Input) The size of the answer area.

Return Value

res_search() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the actual size of the answer returned.

Authorities:

Authorization of *R (allow access to the object) to the host aliases file specified by the *HOSTALIASES* environment variable.

You also need *X authority to each directory in the path of the host aliases file.

Error Conditions

When the *res_search()* function fails, *errno* can be set to one of the following:

[EACCES]

Permission denied. The process does not have the appropriate privileges to the host aliases file specified by the *HOSTALIASES* environment variable.

[EFAULT]

The system detected a pointer that was invalid while attempting to access an input pointer.

[EINVAL]

The **_res** appears to be initialized but the reserved field is not set to zeros.

When the *res_search()* function fails, *h_errno* (defined in **<netdb.h>**) can be set to one of the following:

[HOST_NOT_FOUND]

(Set by the call to *res_query()*) The domain name specified by the *domain_name* parameter was not found. The return code in the response packet was NXDOMAIN.

[TRY_AGAIN]

Either the name server is not running or the name server returned SERVFAIL in the response packet.

[NO_RECOVERY]

(Set by the call to *res_query()*) An unrecoverable error has occurred. Either the domain name could not be compressed because it was invalid or the name server returned FORMERR, NOTIMP, or REFUSED.

[NO_DATA]

(Set by the call to *res_query()*) The domain name exists but there is no data of the requested type.

Usage Notes

1. The *res_search()* function implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. *res_search()* takes the domain name received in *domain_name*, and makes it fully qualified (if it is not already). *res_search()* also calls *res_query()*, passing it the different domain names to look up, until a successful response is received.
2. *res_search()* calls *res_init()* if the **_res** structure has not been initialized.
3. *res_search()* expects EBCDIC data as input. The output from *res_search()* is also EBCDIC.
4. In a thread-enabled environment, the **_res** structure is shared among all threads within a process.
5. *res_search()* will resolve local host aliases to a domain name which are then resolved with a query using DNS. See "res_hostalias()—Retrieve the host alias" on page 291 for more information on aliases.

Related Information

- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_mkquery()—Place Domain Query in Buffer" on page 296—Place Domain Query in Buffer
- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_close()—Close Socket and Reset _res Structure" on page 287Close Socket and Reset _res Structure
- "res_send()—Send Buffered Domain Query or Update"—Send Buffered Domain Query
- "res_xlate()—Translate DNS Packets" on page 323—Translate DNS Packets

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

---

res_send()—Send Buffered Domain Query or Update

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send(char *query_buffer,
             int  query_buffer_length,
             char *answer_buffer,
             int  answer_buffer_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_send()* function is used to send a query or update message to a name server and retrieve a response.

Parameters

query_buffer
(Input) The pointer to the query or update message.

query_buffer_length
(Input) The length of *query_buffer*.

answer_buffer
(Output) The pointer to where the response is stored.

answer_buffer_length
(Input) The size of the answer_buffer.

Authorities:

No authorization is required.

Return Value

res_send() returns an integer. Possible values are:

- -1 (unsuccessful)
- n (successful), where n is the actual size of the answer returned.

Error Conditions

When the *res_send()* function fails, *res_send()* can set *errno* to one of the following:

[ECONNREFUSED]

Not able to connect to a server.

[ECONVERT]

Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

[EINVAL]

One of the following reasons:

An invalid length or NULL pointer was passed to *res_send()* or The **_res** could not be initialized properly or The **_res** appears to be initialized but the reserved field is not set to zeros.

Note: No attempt is made to initialize the **_res** structure if it was initialized previous to the *res_send()* being issued.

[ESRCH]

No DNS servers were specified in *nsaddr*.

[ETIMEDOUT]

A timeout received from a connected server.

When the *res_send()* function fails, *h_errno* (defined in **<netdb.h>**) can also be set to one of the following:

HOST_NOT_FOUND

Either the input packet could not be translated to ASCII or the answer received could not be translated to the coded character set identifier (CCSID) currently in effect for the job.

NO_RECOVERY

An invalid length or NULL pointer was passed to *res_send()* or the **_res** could not be initialized properly.

Notes:

- No attempt is made to initialize the **_res** structure if it was initialized previous to the *res_send()* being issued.
- There are numerous other values that *errno* can be set to by the sockets functions that *res_send()* calls. The above values are the only values that *res_send()* can specifically set. Refer to other sockets functions for the other values. *errno* is always set in an error condition, but *h_errno* is not necessarily set.

After receiving an error reply packet, *res_send()* will set the *extended_error* field in the *state* structure to the last reply return code from the DNS server. See <arpa/nameser.h> for all possible values of *ns_rcode*.

Usage Notes

1. *res_send()* sends the query or update to the local name server and handles all timeouts and retries. The response packet is stored in *answer_buffer*.
2. *res_send()* calls *res_init()* if the **_res** structure has not been initialized.
3. *res_send()* uses the UDP protocol, except for the following cases in which it uses TCP to send the packet.
   - If the RES_USEVC or RES_STAYOPEN bits are set in the options field of the **_res** structure.
   - If the configuration from Change TCP/IP Domain (CHGTCPDMN) specifies that the server protocol is TCP.
   - If the truncation bit is set in the packet header on the response from a UDP packet, and RES_IGNTC is not set in the **_res** structure.
4. *res_send()* does not perform interactive queries and expects the name server to handle recursion.
5. *res_send()* assumes that the data passed to it is EBCDIC and is in the default coded character set identifier (CCSID) currently in effect for the job. It translates the data from the default CCSID currently in effect for the job to ASCII (CCSID 819) before the data is sent out to a name server. The response that it receives from the name server is returned in the default CCSID currently in effect for the job.
6. Unless RES_NOCACHE was specified, *res_send()* checks the cached data for the answer to the query (but not for updates). If the answer is found and the time to live has not expired, it is returned to the calling program in *answer_buffer* and no attempt is made to send it on the network. If the time to live has expired, the entry is deleted from the cache, and the query is sent on the network. If the answer is not found in the cache, *res_send()* also sends the query on the network. When an answer is received from the network, it is placed in cache if it is an authoritative answer and is not the result of an inverse query. RES_NOCACHE does not stop answers from being cached. Authoritative negative replies, indicating the data does not exist, will also be cached.
7. In a thread-enabled environment, the **_res** structure is shared among all threads within a process.

Related Information

- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_mkquery()—Place Domain Query in Buffer" on page 296—Place Domain Query in Buffer
- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_search()—Search for Domain Name" on page 318—Search for Domain Name

- “res_close()—Close Socket and Reset _res Structure” on page 287—Close Socket and Reset _res Structure
- “res_xlate()—Translate DNS Packets”—Translate DNS Packets

API introduced: V3R1

---

res_xlate()—Translate DNS Packets

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_xlate(int  input_ccsid,
              char *input_packet,
              int  input_length,
              int  output_ccsid,
              char *output_packet,
              int  output_length)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *res_xlate()* function is used to translate a standard DNS packet between ASCII and EBCDIC.

Parameters

input_ccsid
(Input) The CCSID value of the input packet to be translated.

input_packet
(Input) The pointer to where the standard DNS packet to be translated resides.

input_length
(Input) The length of *input_packet*.

output_ccsid
(Input) The CCSID value for the output packet.

output_packet
(Output) The pointer to where the translated DNS packet will be stored.

output_length
(Input) The length of *output_packet*.

Authorities

No authorization is required.

Return Value

res_xlate() returns an integer. Possible values are:

- 1 (successful)
- 0 (unsuccessful - translation error)
- -1 (unsuccessful - errors other than translation)

Error Conditions

When the *res_xlate()* function fails, it does not set specific *errno* or *h_errno* values. An error occurs under the following conditions:

- NULL pointer(s) passed to the function.
- Invalid pointer(s) passed to the function.
- Invalid lengths passed to the function.
- An invalid packet format encountered.

Usage Notes

1. *res_xlate()* parses through *input_packet*, determining which fields need translation. The packet is copied into *output_packet* as it is parsed, translating the fields as needed from *input_ccsid* to *output_ccsid*. If a bad format is encountered or a user-supplied length is too small, *res_xlate()* returns a -1.
2. If there is an error in the translation of ***input_packet*** from ***input_ccsid*** to ***output_ccsid***, *res_xlate()* returns a value of 0 to the caller.
3. *res_xlate()* expects a value of 819 (ASCII) for either the input or output coded character set identifier (CCSID). If translation from an EBCDIC CCSID is to occur, the output CCSID needs to be set to 819. ***input_packet*** is then translated to ASCII, and the result is placed in ***output_packet*** If translation to an EBCDIC CCSID is to occur, the input CCSID needs to be set to 819. ***input_packet*** is then translated from ASCII to the EBCDIC CCSID specified in ***output_ccsid***, and the result is placed in ***output_packet***.

   res_xlate() returns unsuccessfully with a value of -1 if CCSID 819 is not used for either ***input_ccsid*** or ***output_ccsid***. Also, if both ***input_ccsid*** and ***output_ccsid*** values are 819, *res_xlate()* returns a -1.
4. In a thread-enabled environment, the **_res** is shared among all threads within a process.

Related Information

- "hstrerror()—Retrieve Resolver Error Message" on page 267—Retrieve Resolver Error Message
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_init()—Initialize _res Structure" on page 292—Initialize _res Structure
- "res_mkquery()—Place Domain Query in Buffer" on page 296—Place Domain Query in Buffer
- "res_query()—Send Domain Query" on page 316—Send Domain Query
- "res_search()—Search for Domain Name" on page 318—Search for Domain Name
- "res_send()—Send Buffered Domain Query or Update" on page 320—Send Buffered Domain Query
- "res_close()—Close Socket and Reset _res Structure" on page 287—Close Socket and Reset _res Structure
- "res_findzonecut()—Find the Enclosing Zone and Servers" on page 288—Find the Enclosing Zone and Servers
- "res_hostalias()—Retrieve the host alias" on page 291—Retrieve the host alias
- "res_ninit()—Initialize res Structure" on page 299—Initialize res Structure
- "res_nclose()—Close Socket and Reset res Structure" on page 299—Close Socket and Reset res Structure
- "res_nmkquery()—Place Domain Query in Buffer" on page 305—Place Domain Query in Buffer
- "res_nmkupdate()—Construct an Update Packet" on page 306—Construct an Update Packet
- "res_nquery()—Send Domain Query" on page 307—Send Domain Query
- "res_nsearch()—Search for Domain Name" on page 309—Search for Domain Name
- "res_nsend()—Send Buffered Domain Query or Update" on page 310—Send Buffered Domain Query
- "res_nsendsigned()—Send Authenticated Domain Query or Update" on page 311—Send Authenticated Domain Query
- "res_nupdate()—Build and Send Dynamic Updates" on page 314—Build and Send Dynamic Updates

API introduced: V3R1

sethostent()—Open Host Database

Syntax

```
#include <netdb.h>

void sethostent(int stay_open)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *sethostent()* function is used to prepare for sequential access to the host database file. *sethostent()* opens the file and repositions the file marker to the beginning of the file. In addition, *sethostent()* affects what type of transport service (connectionless versus connection-oriented) is to be used when *gethostbyname()* and *gethostbyaddr()* need to retrieve host information from the domain name server.

Parameters

int stay_open

(Input) Specifies whether to leave the database file open after each call to *gethostbyname()* and *gethostbyaddr()*. A nonzero value results in the database file being left open. Also, a nonzero value results in the use of a connection-oriented transport service (for example, TCP) being used by *gethostbyname()* and *gethostbyaddr()* when host information is to be obtained from the domain name server.

Authorities

No authorization is required.

Error Conditions

When *sethostent()* fails, *h_errno* (defined in **<netdb.h>**) can be set to one of the following:

NO_RECOVERY

An unrecoverable error has occurred.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the host database file:
   - ADDTCPHTE (Add TCP/IP Host Table Entry
   - RMVTCPHTE (Remove TCP/IP Host Table Entry)
   - CHGTCPHTE (Change TCP/IP Host Table Entry)
   - RNMTCPHTE (Rename TCP/IP Host Table Entry)
   - MRGTCPHT (Merge TCP/IP Host Tables)
2. Do not use the *sethostent()* function in a multithreaded environment. See the multithread alternative *sethostent_r()* function.
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *sethostent()* API is mapped to*qso_sethostent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface

- "gethostbyaddr()—Get Host Information for IP Address" on page 222—Get Host Information for IP Address
- "gethostbyname()—Get Host Information for Host Name" on page 227—Get Host Information for Host Name
- "endhostent()—Close Host Database" on page 206—Close Host Database
- "gethostent()—Get Next Entry from Host Database" on page 233—Get Next Entry from Host Database

API introduced: V3R1

---

sethostent_r()—Open Host Database

Syntax

```
#include <netdb.h>

int sethostent_r(int stay_open,
       struct hostent_data *hostent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *sethostent_r()* function is used in preparation for sequential access to the host database file. The *sethostent_r()* function opens the file and repositions the file marker to the beginning of the file. In addition, this call affects what type of transport service (connectionless versus connection-oriented) that is to be used when *gethostbyname_r()* and *gethostbyaddr_r()* need to retrieve host information from the domain name server.

Parameters

int stay_open (input)
Specifies whether to leave the database file open after each call to *gethostbyname_r()* and *gethostbyaddr_r()*. A non-zero value will result in the database file being left open. Also, a non-zero value will result in the use of a connection-oriented transport service (for example, TCP) being used by *gethostbyname_r()* and *gethostbyaddr_r()* when host information is to be obtained from the domain name server.

struct hostent_data *hostent_data_struct_addr (input/output)
Specifies the pointer to the hostent_data structure, which is used to pass and preserve results between function calls. The field host_control_blk in the hostent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire hostent_data structure must be initialized to hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *sethostent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct hostent_data**denoted by ***hostent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *sethostent_r()* function fails, *h_errno* (defined in **<netdb.h>**) can be set to:

[NO_RECOVERY]

An unrecoverable error has occurred.

When the *sethostent_r()* function fails, *errno* can be set to:

[EINVAL]

The hostent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure hostent_data.

Usage Notes

The iSeries Navigator or the following CL commands can be used to access the host database file:

- ADDTCPHTE (Add TCP/IP Host Table Entry)
- RMVTCPHTE (Remove TCP/IP Host Table Entry)
- CHGTCPHTE (Change TCP/IP Host Table Entry)
- RNMTCPHTE (Rename TCP/IP Host Table Entry)
- MRGTCPHT (Merge TCP/IP Host Tables)

Related Information

- "gethostbyaddr_r()—Get Host Information for IP Address" on page 224—Get Host Information for IP Address
- "gethostbyname_r()—Get Host Information for Host Name" on page 230—Get Host Information for Host Name
- "endhostent_r()—Close Host Database" on page 207—Close Host Database
- "gethostent_r()—Get Next Entry from Host Database" on page 235—Get Next Entry from Host Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

setnetent()—Open Network Database

Syntax

```
#include <netdb.h>

void setnetent(int stay_open)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 328.

The *setnetent()* function is used to prepare for sequential access to the network database file. *setnetent()* opens the file and repositions the file marker to the beginning of the file.

Parameters

stay_open

(Input) A value that indicates whether to leave the database file open after each *getnetbyname()* and *getnetbyaddr()*. A nonzero value will result in the database file being left open.

Authorities

No authorization is required.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the network database file:
   - WRKNETTBLE (Work with Network Table Entries)
   - ADDNETTBLE (Add Network Table Entry)
   - RMVNETTBLE (Remove Network Table Entry)
2. Do not use the *setnetent()* function in a multithreaded environment. See the multithread alternative *setnetent_r()* function.
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *setnetent()* API is mapped to *qso_setnetent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getnetbyaddr()—Get Network Information for IP Address" on page 239—Get Network Information for IP Address
- "getnetbyname()—Get Network Information for Domain Name" on page 242—Get Network Information for Domain Name
- "getnetent()—Get Next Entry from Network Database" on page 245—Get Next Entry from Network Database
- "endnetent()—Close Network Database" on page 209—Close Network Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

setnetent_r()—Open Network Database

Syntax

```
#include <netdb.h>

int setnetent_r(int stay_open,
            struct netent_data
               *netent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *setnetent_r()* function is used in preparation for sequential access to the network database file. The *setnetent_r()* function opens the file and repositions the file marker to the beginning of the file.

Parameters

int stay_open (input)
Specifies whether to leave the database file open after each call to *getnetbyname_r()* and *getnetbyaddr_r()*. A non-zero value will result in the database file being left open.

struct netent_data *netent_data_struct_addr (input/output)
Specifies the pointer to the netent_data structure, which is used to pass and preserve results between function calls. The field net_control_blk in the netent_data structure must be initialized

with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire netent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *setnetent_r()* function returns a pointer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct netent_data**denoted by ***netent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *setnetent_r()* function fails, *errno* can be set to:

[EINVAL]

The netent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action see the description for structure netent_data.

Usage Notes

The iSeries Navigator or the following CL commands can be used to access the network database file:

- WRKNETTBLE (Work with Network Table Entries)
- ADDNETTBLE (Add Network Table Entry)
- RMVNETTBLE (Remove Network Table Entry)

Related Information

- "getnetent_r()—Get Next Entry from Network Database" on page 246—Get Next Entry from Network Database
- "getnetbyaddr_r()—Get Network Information for IP Address" on page 240—Get Network Information for IP Address
- "getnetbyname_r()—Get Network Information for Domain Name" on page 244—Get Network Information for Domain Name
- "endnetent_r()—Close Network Database" on page 210—Close Network Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

setprotoent()—Open Protocol Database

Syntax

```
#include <netdb.h>

void setprotoent(int stay_open)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes" on page 330.

The *setprotoent()* function is used to prepare for sequential access to the protocol database file. *setprotoent()* opens the file and repositions the file marker to the beginning of the file.

Parameters

stay_open
(Input) A value that indicates whether to leave the database file open after each *getprotobynumber()* and *getprotobyname()*. A nonzero value results in the database file being left open.

Authorities

No authorization is required.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the protocol database file:

   - WRKPCLTBLE (Work with Protocol Table Entries)
   - ADDPCLTBLE (Add Protocol Table Entry)
   - RMVPCLTBLE (Remove Protocol Table Entry)
2. Do not use the *setprotoent()* function in a multithreaded environment. See the multithread alternative *setprotoent_r()* function.
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *setprotoent()* API is mapped to *qso_setprotoent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getprotobyname()—Get Protocol Information for Protocol Name" on page 248—Get Protocol Information for Protocol Name
- "getprotobynumber()—Get Protocol Information for Protocol Number" on page 251—Get Protocol Information for Protocol Number
- "getprotoent()—Get Next Entry from Protocol Database" on page 254—Get Next Entry from Protocol Database
- "endprotoent()—Close Protocol Database" on page 211—Close Protocol Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

setprotoent_r()—Open Protocol Database

Syntax

```
#include <netdb.h>

int setprotoent_r(int stay_open,
   struct protoent_data *protoent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *setprotoent_r()* function is used in preparation for sequential access to the protocol database file. The *setprotoent_r()* function opens the file and repositions the file marker to the beginning of the file.

Parameters

int stay_open (input)
Specifies whether to leave the database file open after each call to *getprotobynumber_r()* and *getprotobyname_r()*. A non-zero value will result in the database file being left open.

struct protoent_data *protoent_data_struct_addr (input/output)
Specifies the pointer to the protoent_data structure, which is used to pass and preserve results between function calls. The field proto_control_blk in the protoent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire protoent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *setprotoent_r()* returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct protoent_data** denoted by ***protoent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *setprotoent_r()* function fails, *errno* can be set to:

[EINVAL]

The protoent_data structure was not properly initialized with hexadecimal zeros before initial use. For corrective action, see the description for structure protoent_data.

Usage Notes

The iSeries Navigator or the following CL commands can be used to access the protocol database file:

- WRKPCLTBLE (Work with Protocol Table Entries)
- ADDPCLTBLE (Add Protocol Table Entry)
- RMVPCLTBLE (Remove Protocol Table Entry)

Related Information

- “getprotobynumber_r()—Get Protocol Information for Protocol Number” on page 253-Get Protocol
- “getprotobyname_r()—Get Protocol Information for Protocol Name” on page 250—Get Protocol Information for Protocol Name
- “endprotoent_r()—Close Protocol Database” on page 212—Close Protocol Database
- “getprotoent_r()—Get Next Entry from Protocol Database” on page 255—Get Next Entry from Protocol Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

setservent()—Open Service Database

Syntax

```
#include <netdb.h>

void setservent(int stay_open)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: No; see "Usage Notes."

The *setservent()* function is used to prepare for sequential access to the service database file. *setservent()* opens the file and repositions the file marker to the beginning of the file.

Parameters

stay_open
(Input) A value that indicates whether to leave the database file open after each *getservbyname()* and *getservbyport()*. A nonzero value results in the database file being left open.

Authorities

No authorization is required.

Usage Notes

1. The iSeries Navigator or the following CL commands can be used to access the services database file:
   - WRKSRVTBLE (Work with Service Table Entries)
   - ADDSRVTBLE (Add Service Table Entry)
   - RMVSRVTBLE (Remove Service Table Entry)
2. Do not use the *setservent()* function in a multithreaded environment. See the multithread alternative *setservent_r()* function.
3. When you develop in C-based languages and an application is compiled with the _XOPEN_SOURCE macro defined to the value 520 or greater, the *setservent()* API is mapped to *qso_setservent98()*.

Related Information

- _XOPEN_SOURCE—Using _XOPEN_SOURCE for the UNIX 98 compatible interface
- "getservbyname()—Get Port Number for Service Name" on page 257—Get Port Number for Service Name
- "getservbyport()—Get Service Name for Port Number" on page 261—Get Service Name for Port Number
- "getservent()—Get Next Entry from Service Database" on page 264—Get Next Entry from Service Database
- "endservent()—Close Service Database" on page 213—Close Service Database

API introduced: V4R2

Top | UNIX-Type APIs | APIs by category

---

setservent_r()—Open Service Database

Syntax

```
#include <netdb.h>

int setservent_r(int stay_open,
      struct servent_data *servent_data_struct_addr)
```

Service Program Name: QSOSRV2
Default Public Authority: *USE
Threadsafe: Yes

The *setservent_r()* function is used in preparation for sequential access to the service database file. The *setservent_r()* function opens the file and repositions the file marker to the beginning of the file.

Parameters

int stay_open (input)
Specifies whether to leave the database file open after each call to *getservbyname_r()* and *getservbyport_r()*. A non-zero value will result in the database file being left open.

struct servent_data *servent_data_struct_addr (input/output)
Specifies the pointer to the servent_data structure, which is used to pass and preserve results between function calls. The field serve_control_blk in the servent_data structure must be initialized with hexadecimal zeros before its initial use. If compatibility with other platforms is required, then the entire servent_data structure must be initialized with hexadecimal zeros before initial use.

Authorities

No authorization is required.

Return Value

The *setservent_r()* function returns an integer. Possible values are:

- -1 (unsuccessful call)
- 0 (successful call)

The **struct servent_data**denoted by ***servent_data_struct_addr*** is defined in **<netdb.h>**.

Error Conditions

When the *setservent_r()* function fails, *errno* can be set to:

[EINVAL]

The servent_data structure was not properly initialized to hexadecimal zeros before initial use. For corrective action, see the description for structure servent_data.

Usage Notes

The iSeries Navigator or the following CL commands can be used to access the services database file:

- WRKSRVTBLE (Work with Service Table Entries)
- ADDSRVTBLE (Add Service Table Entry)
- RMVSRVTBLE (Remove Service Table Entry)

Related Information

- “getservbyname_r()—Get Port Number for Service Name” on page 259—Get Port Number for Service Name
- “getservbyport_r()—Get Service Name for Port Number” on page 262—Get Service Name for Port Number
- “endservent_r()—Close Service Database” on page 214—Close Service Database
- “getservent_r()—Get Next Entry from Service Database” on page 265—Get Next Entry from Service Database

API introduced: V4R2

_getlong()—Get Long Byte Quantities

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

unsigned long
         _getlong(unsigned char *message_pointer)
```

Threadsafe: Yes

The *_getlong()* function is used to retrieve an unsigned long byte quantity.

Authorities and Locks

None.

Parameters

message_pointer
(Input) The pointer where the long integer is to be received from.

Return Value

_getlong() returns a 32-bit integer from where *message_pointer* is pointing.

Usage Notes

1. DNS packets have fields that are unsigned long integers (for example, TTL and serial number). *_getlong()* picks these unsigned long integers out of a DNS packet and returns them.

Related Information

- "_getshort()—Get Short Byte Quantities"—Get Short Byte Quantities
- "_putlong()—Put Long Byte Quantities" on page 335—Put Long Byte Quantities
- "_putshort()—Put Short Byte Quantities" on page 336—Put Short Byte Quantities

API introduced: V3R1

_getshort()—Get Short Byte Quantities

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

unsigned short
         _getshort(unsigned char *message_pointer)
```

Threadsafe: Yes

The *_getshort()* function is used to retrieve an unsigned short byte quantity.

Authorities and Locks

None.

Parameters

message_pointer
(Input) The pointer where the short integer is to be received from.

Return Value

_getshort() returns a 16-bit integer from where *message_pointer* is pointing.

Usage Notes

1. DNS packets have fields that are unsigned short integers (for example, type, class, and data length). *_getshort()* picks these unsigned short integers out of a DNS packet and returns them.

Related Information

- "_getlong()—Get Long Byte Quantities" on page 334—Get Long Byte Quantities
- "_putlong()—Put Long Byte Quantities"—Put Long Byte Quantities
- "_putshort()—Put Short Byte Quantities" on page 336—Put Short Byte Quantities

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

_putlong()—Put Long Byte Quantities

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void  _putlong(unsigned long long_integer,
               unsigned char *message_pointer)
```

Threadsafe: Yes

The *_putlong()* function is used to put an unsigned long byte quantity into a byte stream.

Authorities and Locks

None.

Parameters

long_int
(Input) The 32-bit integer to be put into the byte stream.

unsigned char *message_pointer
(Input) The pointer to where the *long_integer* is to be put.

Return Value

_putlong() puts a 32-bit integer into *message_pointer*.

Usage Notes

DNS packets have fields that are unsigned long integers (for example, TTL and serial number). *_putlong()* is generally used to put these fields into a DNS packet.

Related Information

- "_getlong()—Get Long Byte Quantities" on page 334—Get Long Byte Quantities
- "_getshort()—Get Short Byte Quantities" on page 334—Get Short Byte Quantities
- "_putshort()—Put Short Byte Quantities"—Put Short Byte Quantities

API introduced: V3R1

Top | UNIX-Type APIs | APIs by category

_putshort()—Put Short Byte Quantities

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void  _putshort(unsigned short short_integer,
                unsigned char  *message_pointer)
```

Threadsafe: Yes

The *_putshort()* function is used to put an unsigned short byte quantity into a byte stream.

Authorities and Locks

None.

Parameters

unsigned short short_int
(Input) The 16-bit integer to be put into the byte stream.

unsigned char *message_pointer
(Input) The pointer to where the *short_integer* is to be put.

Return Value

_putshort() puts a 16-bit integer into *message_pointer*.

Usage Notes

DNS packets have fields that are unsigned short integers (for example, type, class, and data length). *_putshort()* is generally used to put these fields into a DNS packet.

Related Information

- "_getlong()—Get Long Byte Quantities" on page 334—Get Long Byte Quantities
- "_getshort()—Get Short Byte Quantities" on page 334—Get Short Byte Quantities

- "_putlong()—Put Long Byte Quantities" on page 335—Put Long Byte Quantities

API introduced: V3R1

Concepts

These are the concepts for this category.

Debugging IP over SNA Configurations

Two commands can be helpful in debugging IP over SNA configurations:

- The Start Mode (STRMOD) CL command can help you determine if your SNA configuration is correct. As input to the STRMOD command, you need the remote location name. You can determine the remote location name from the destination IP address by using the Convert IP over SNA Interface (CVTIPSIFC) command. The message you receive when STRMOD completes tells you whether it was successful.
- The TCP/IP FTP command can help you determine if your AnyNet configuration is correct. If you get the *User* prompt, the AnyNet configuration is correct.

  Note: When FTP fails, it does not give a detailed reason for the failure. To get a detailed reason, you should run a sockets program that reports the value for *errno* when the failure occurs.

Common IP over SNA Configuration Errors

Sockets Error (value of *errno*)	**Possible Causes**
EHOSTUNREACH	1. Missing ADDIPSLOC command on client system. 2. Missing ADDIPSIFC command on client system. 3. Type of service points to a non-existent mode description on client system. 4. ADDIPSLOC command on client system resulted in a location name that is not found. 5. ADDIPSLOC command on client system resulted in a location name that is on a non-APPC device description.
EADDRNOTAVAIL	1. AnyNet not active on client system (ALWANYNET attribute set to *NO), but TCP is started. 2. Mode could not be added to device on client system.
EUNATCH	1. AnyNet not active on client system (ALWANYNET attribute set to *NO), and TCP is not started.
ECONNREFUSED	1. AnyNet not active on client system (ALWANYNET attribute set to *NO). 2. *listen()* not active on server system.

Sockets Error (value of *errno*)	**Possible Causes**
ECONNABORTED	1. Line error 2. Device/controller/line varied off on client or server system while in use. 3. User not authorized to APPC device description object on server system.
ETIMEDOUT	1. ADDIPSLOC command on client system points to a location name that does not exist or is on a system that is not responding in the APPN network. 2. Messages (especially inquiry messages) on message queue QSYSOPR are waiting for a reply.
EACCES	1. User not authorized to port on client system. 2. User not authorized to APPC device description object on client system.

Top | UNIX-Type APIs | APIs by category

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

```
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY  10504-1785
U.S.A.
```

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

```
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan
```

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This Application Programming Interfaces (API) publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:
Advanced 36
Advanced Function Printing
Advanced Peer-to-Peer Networking
AFP
AIX
AS/400
COBOL/400
CUA
DB2
DB2 Universal Database
Distributed Relational Database Architecture
Domino
DPI
DRDA
eServer
GDDM
IBM
Integrated Language Environment
Intelligent Printer Data Stream
IPDS
i5/OS
iSeries
Lotus Notes
MVS
Netfinity
Net.Data
NetView
Notes
OfficeVision
Operating System/2
Operating System/400
OS/2
OS/400
PartnerWorld
PowerPC
PrintManager
Print Services Facility
RISC System/6000
RPG/400
RS/6000
SAA
SecureWay
System/36
System/370
System/38
System/390
VisualAge
WebSphere
xSeries

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

Other company, product, and service names may be trademarks or service marks of others.

Terms and Conditions

Permissions for the use of these Publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these Publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these Publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations. IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE

Printed in USA