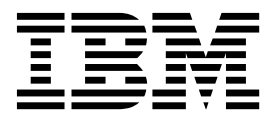


IBM i
Version 7.2

*Programming
Socket programming*



IBM i
Version 7.2

*Programming
Socket programming*



Note

Before using this information and the product it supports, read the information in “Notices” on page 195.

This edition applies to version IBM i 7.2 (product number 5770-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.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

© **Copyright IBM Corporation 2001, 2013.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Socket programming 1

What's new for IBM i 7.2	1
PDF files for Socket programming	1
Prerequisites for socket programming	3
How sockets work	3
Socket characteristics	6
Socket address structure	7
Socket address family	8
AF_INET address family	8
AF_INET6 address family	9
AF_UNIX address family	10
AF_UNIX_CCSID address family	11
Socket type	12
Socket protocols	13
Basic socket design	13
Creating a connection-oriented socket	13
Example: A connection-oriented server	15
Example: A connection-oriented client	19
Creating a connectionless socket	22
Example: A connectionless server	23
Example: A connectionless client	25
Designing applications with address families	27
Using AF_INET address family	27
Using AF_INET6 address family	28
Using AF_UNIX address family	29
Example: Server application that uses	
AF_UNIX address family	31
Example: Client application that uses	
AF_UNIX address family	34
Using AF_UNIX_CCSID address family	36
Example: Server application that uses	
AF_UNIX_CCSID address family	38
Example: Client application that uses	
AF_UNIX_CCSID address family	41
Advanced socket concepts	43
Asynchronous I/O	43
Secure sockets	46
Global Security Kit (GSKit) APIs	47
SSL APIs	50
Secure socket API error code messages	51
Client SOCKS support	54
Thread safety	58
Nonblocking I/O	58
Signals	60
IP multicasting	61
File data transfer—send_file() and	
accept_and_recv()	62
Out-of-band data	63
I/O multiplexing—select()	64
Socket network functions	64
Domain Name System support	65
Environment variables	66
Data caching	67
Domain Name System Security Extensions	
(DNSSEC)	67
Berkeley Software Distribution compatibility	68

UNIX 98 compatibility	71
Descriptor passing between processes: sendmsg()	
and recvmsg()	74
Sockets-related User Exit Points	76
Example: User Exit Program for	
QIBM_QSO_ACCEPT	77
Socket scenario: Creating an application to accept	
IPv4 and IPv6 clients	79
Example: Accepting connections from both IPv6	
and IPv4 clients	80
Example: IPv4 or IPv6 client	85
Socket application design recommendations	88
Examples: Socket application designs	91
Examples: Connection-oriented designs	91
Example: Writing an iterative server program	92
Example: Using the spawn() API to create	
child processes	96
Example: Creating a server that uses	
spawn()	98
Example: Enabling the worker job to	
receive a data buffer	100
Example: Passing descriptors between	
processes	101
Example: Server program used for	
sendmsg() and recvmsg()	103
Example: Worker program used for	
sendmsg() and recvmsg()	107
Examples: Using multiple accept() APIs to	
handle incoming requests	108
Example: Server program to create a pool	
of multiple accept() worker jobs	110
Example: Worker jobs for multiple accept()	112
Example: Generic client	113
Example: Using asynchronous I/O	116
Examples: Establishing secure connections	123
Example: GSKit secure server with	
asynchronous data receive	123
Example: GSKit secure server with	
asynchronous handshake	133
Example: Establishing a secure client with	
Global Security Kit APIs	143
Example: Using gethostbyaddr_r() for threadsafe	
network routines	150
Example: Nonblocking I/O and select()	152
Using poll() instead of select()	158
Example: Using signals with blocking socket	
APIs	164
Examples: Using multicasting with AF_INET	167
Example: Sending multicast datagrams	169
Example: Receiving multicast datagrams	171
Example: Updating and querying DNS	172
Examples: Transferring file data using	
send_file() and accept_and_recv() APIs	176
Example: Using accept_and_recv() and	
send_file() APIs to send contents of a file	177
Example: Client request for a file	181

Xsockets tool	183
Configuring Xsockets.	183
What is created by integrated Xsocket setup	184
Configuring Xsockets to use a Web browser . .	186
Configuring an Integrated Web Application	
Server.	186
Updating configuration files	187
Configuring Xsockets Web application . .	188
Testing Xsockets tool in a Web browser. .	189
Using Xsockets	189
Using integrated Xsockets	189

Using Xsockets in a Web browser.	190
Deleting objects created by the Xsockets tool	191
Customizing Xsockets	191
Serviceability tools	192

Notices 195

Programming interface information	197
Trademarks	197
Terms and conditions.	197

Socket programming

A *socket* is a communications connection point (endpoint) that you can name and address in a network. Socket programming shows how to use socket APIs to establish communication links between remote and local processes.

The processes that use a socket can reside on the same system or different systems on different networks. Sockets are useful for both stand-alone and network applications. Sockets allow you to exchange information between processes on the same machine or across a network, distribute work to the most efficient machine, and they easily allow access to centralized data. Socket application program interfaces (APIs) are the network standard for TCP/IP. A wide range of operating systems support socket APIs. IBM® i sockets support multiple transport and networking protocols. Socket system functions and the socket network functions are threadsafe.

Programmers who use Integrated Language Environment® (ILE) C can refer to this topic collection to develop socket applications. You can also code to the sockets API from other ILE languages, such as RPG.

The Java™ language also supports a socket programming interface.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

What's new for IBM i 7.2



Read about new or significantly changed information for the Socket Programming Guide.

Domain Name System Security Extensions (DNSSEC)

The resolver APIs added support for the “Domain Name System Security Extensions (DNSSEC)” on page 67.

How to see what's new or changed

To help you see where technical changes have been made, the information center uses:

- The  image to mark where new or changed information begins.
- The  image to mark where new or changed information ends.

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF files for Socket programming



You can view and print a PDF file of this information.

To view or download the PDF version of this document, select Socket programming.




Other information

You can also view or print any of the following PDFs:

IBM Redbooks®:

- Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More  (5630 KB)
- IBM eServer™ iSeries Wired Network Security: OS/400® V5R1 DCM and Cryptographic Enhancements  (10 035 KB)

You can view or download these related topics:


- **IPv6**
 - RFC 3493: "Basic Socket Interface Extensions for IPv6" 
 - RFC 3513: "Internet Protocol Version 6 (IPv6) Addressing Architecture" 
 - RFC 3542: "Advanced Sockets Application Program Interface (API) for IPv6" 
- **Domain Name System**
 - RFC 1034: "Domain Names - Concepts and Facilities" 
 - RFC 1035: "Domain Names - Implementation and Specification" 
 - RFC 2136: "Dynamic Updates in the Domain Name System (DNS UPDATE)" 
 - RFC 2181: "Clarifications to the DNS Specification" 
 - RFC 2308: "Negative Caching of DNS Queries (DNS NCACHE)" 
 - RFC 2845: "Secret Key Transaction Authentication for DNS (TSIG)" 
- **Secure Sockets Layer/Transport Layer Security**
 - RFC 5246: "The Transport Layer Security (TLS) Protocol Version 1.2" 
 - RFC 4346: "The Transport Layer Security (TLS) Protocol Version 1.1" 
 - RFC 2246: "The TLS Protocol Version 1.0 " 
 - RFC 2560: "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP" 
- **Other Web Resources**
 - Technical Standard: Networking Services (XNS), Issue 5.2 Draft 2.0 

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html) .

Prerequisites for socket programming

Before writing socket applications, you must complete these steps to meet the requirements for compiler, AF_INET and AF_INET6 address families, Secure Sockets Layer (SSL) APIs, and Global Security Kit (GSKit) APIs.

Compiler requirements

1. Install QSYSINC library. This library provides necessary header files that are needed when compiling socket applications.
2. Install the ILE C licensed program (5770-WDS Option 51).

Requirements for AF_INET and AF_INET6 address families

In addition to the compiler requirements, you must complete these tasks:

1. Plan TCP/IP setup.
2. Install TCP/IP.
3. Configure TCP/IP for the first time.
4. Configure IPv6 for TCP/IP if you plan to write applications that use the AF_INET6 address family.

Requirements for Secure Sockets Layer (SSL) APIs and Global Security Kit (GSKit) APIs

In addition to the requirements for compiler, AF_INET address families, and AF_INET6 address families, you must complete the following tasks to work with secure sockets:

1. Install and configure Digital Certificate Manager licensed program (5770-SS1 Option 34). See Digital Certificate Manager (DCM) in the information center for details.
2. If you want to use SSL with the cryptographic hardware, you need to install and configure the 4765 Cryptographic Coprocessor. See Cryptography for complete descriptions of the 4765 Cryptographic Coprocessor.

Related reference:

“Using AF_INET address family” on page 27

AF_INET address family sockets can be either connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM). Connection-oriented AF_INET sockets use Transmission Control Protocol (TCP) as the transport protocol. Connectionless AF_INET sockets use User Datagram Protocol (UDP) as the transport protocol.

“Using AF_INET6 address family” on page 28

AF_INET6 sockets provide support for Internet Protocol version 6 (IPv6) 128 bit (16 byte) address structures. Programmers can write applications using the AF_INET6 address family to accept client requests from either IPv4 or IPv6 nodes, or from IPv6 nodes only.

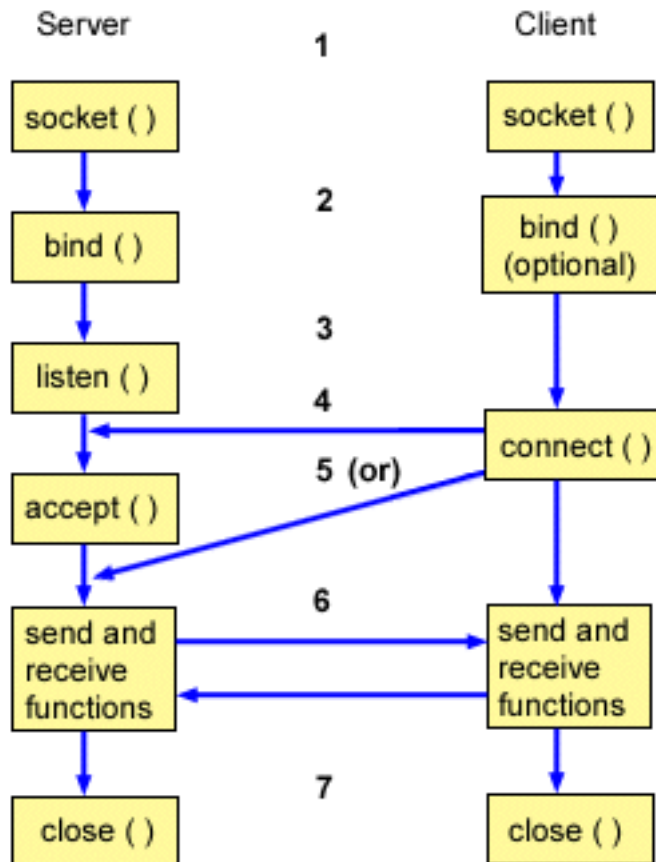
How sockets work

Sockets are commonly used for client and server interaction. Typical system configuration places the server on one machine, with the clients on other machines. The clients connect to the server, exchange information, and then disconnect.

A socket has a typical flow of events. In a connection-oriented client-to-server model, the socket on the server process waits for requests from a client. To do this, the server first establishes (binds) an address that clients can use to find the server. When the address is established, the server waits for clients to request a service. The client-to-server data exchange takes place when a client connects to the server through a socket. The server performs the client's request and sends the reply back to the client.

Note: Currently, IBM supports two versions of most sockets APIs. The default IBM i sockets use Berkeley Socket Distribution (BSD) 4.3 structures and syntax. The other version of sockets uses syntax and structures compatible with BSD 4.4 and the UNIX 98 programming interface specifications. Programmers can specify `_XOPEN_SOURCE` macro to use the UNIX 98 compatible interface.

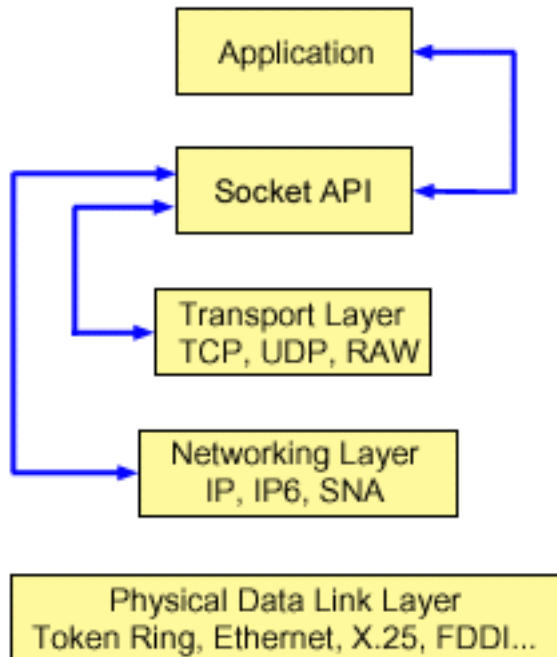
The following figure shows the typical flow of events (and the sequence of issued APIs) for a connection-oriented socket session. An explanation of each event follows the figure.



This is a typical flow of events for a connection-oriented socket:

1. The `socket()` API creates an endpoint for communications and returns a socket descriptor that represents the endpoint.
2. When an application has a socket descriptor, it can bind a unique name to the socket. Servers must bind a name to be accessible from the network.
3. The `listen()` API indicates a willingness to accept client connection requests. When a `listen()` API is issued for a socket, that socket cannot actively initiate connection requests. The `listen()` API is issued after a socket is allocated with a `socket()` API and the `bind()` API binds a name to the socket. A `listen()` API must be issued before an `accept()` API is issued.
4. The client application uses a `connect()` API on a stream socket to establish a connection to the server.
5. The server application uses the `accept()` API to accept a client connection request. The server must issue the `bind()` and `listen()` APIs successfully before it can issue an `accept()` API.
6. When a connection is established between stream sockets (between client and server), you can use any of the socket API data transfer APIs. Clients and servers have many data transfer APIs from which to choose, such as `send()`, `recv()`, `read()`, `write()`, and others.
7. When a server or client wants to stop operations, it issues a `close()` API to release any system resources acquired by the socket.

Note: The socket APIs are located in the communications model between the application layer and the transport layer. The socket APIs are not a layer in the communication model. Socket APIs allow applications to interact with the transport or networking layers of the typical communications model. The arrows in the following figure show the position of a socket, and the communication layer that the socket provides.



Typically, a network configuration does not allow connections between a secure internal network and a less secure external network. However, you can enable sockets to communicate with server programs that run on a system outside a firewall (a very secure host).

Sockets are also a part of IBM's AnyNet® implementation for the Multiprotocol Transport Networking (MPTN) architecture. MPTN architecture provides the ability to operate a transport network over additional transport networks and to connect application programs across transport networks of different types.

Related reference:

"Berkeley Software Distribution compatibility" on page 68
Sockets is a Berkeley Software Distribution (BSD) interface.

"UNIX 98 compatibility" on page 71

Created by The Open Group, a consortium of developers and vendors, UNIX 98 improved the inter-operability of the UNIX operating system while incorporating much of the Internet-related function for which UNIX had become known.

Related information:

socket()--Create Socket API

listen()--Invite Incoming Connections Requests API

bind()--Set Local Address for Socket API

accept()--Wait for Connection Request and Make Connection API

send()--Send Data API

recv()--Receive Data API

close()--Close File or Socket Descriptor API

Sockets APIs

Socket characteristics

Sockets share some common characteristics.

- A socket is represented by an integer. That integer is called a *socket descriptor*.
- A socket exists as long as the process maintains an open link to the socket.
- You can name a socket and use it to communicate with other sockets in a communication domain.
- Sockets perform the communication when the server accepts connections from them, or when it exchanges messages with them.
- You can create sockets in pairs (only for sockets in the AF_UNIX address family).

The connection that a socket provides can be connection-oriented or connectionless. *Connection-oriented* communication implies that a connection is established, and a dialog between the programs follows. The program that provides the service (the server program) establishes the available socket that is enabled to accept incoming connection requests. Optionally, the server can assign a name to the service that it supplies, which allows clients to identify where to obtain and how to connect to that service. The client of the service (the client program) must request the service of the server program. The client does this by connecting to the distinct name or to the attributes associated with the distinct name that the server program has designated. It is similar to dialing a telephone number (an identifier) and making a connection with another party that is offering a service (for example, a plumber). When the receiver of the call (the server, in this example, a plumber) answers the telephone, the connection is established. The plumber verifies that you have reached the correct party, and the connection remains active as long as both parties require it.

Connectionless communication implies that no connection is established, over which a dialog or data transfer can take place. Instead, the server program designates a name that identifies where to reach it (much like a post-office box). If you send a letter to a post office box, you cannot be absolutely sure that the receiver got the letter. You might need to wait for a response to your letter. There is no active, real-time connection, in which data is exchanged.

How socket characteristics are determined

When an application creates a socket with the `socket()` API, it must identify the socket by specifying these parameters:

- The socket address family determines the format of the address structure for the socket. This topic contains examples of each address family's address structure.
- The socket type determines the form of communication for the socket.
- The socket protocol determines the supported protocols that the socket uses.

These parameters or characteristics define the socket application and how it interoperates with other socket applications. Depending on the address family of a socket, you have different choices for the socket type and protocol. The following table shows the corresponding address family and its associated socket type and protocols:

Table 1. Summary of socket characteristics

Address family	Socket type	Socket protocol
AF_UNIX	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP, ICMP

Table 1. Summary of socket characteristics (continued)

Address family	Socket type	Socket protocol
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP6, ICMP6
AF_UNIX_CCSID	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A

In addition to these socket characteristics or parameters, constant values are defined in network routines and header files that are shipped with the QSYSINC library. For descriptions of header files, see the individual APIs. Each API lists its appropriate header file in the usage section of the API description.

Socket network routines allow socket applications to obtain information from the Domain Name System (DNS), host, protocol, service, and network files.

Related reference:

“Socket network functions” on page 64

Socket network functions allow application programs to obtain information from the host, protocol, service, and network files.

Related information:

Sockets APIs

Socket address structure

Sockets use the **sockaddr** address structure to pass and receive addresses. This structure does not require the socket API to recognize the addressing format.

Currently, the IBM i operating system supports Berkeley Software Distribution (BSD) 4.3 and X/Open Single UNIX Specification (UNIX 98). The base IBM i API uses BSD 4.3 structures and syntax. You can select the UNIX 98 compatible interface by defining the `_XOPEN_SOURCE` macro to a value of 520 or greater. Each socket address structure for BSD 4.3 that is used has an equivalent UNIX 98 structure.

Table 2. Comparison of BSD 4.3 and BSD 4.4/ UNIX 98 socket address structure

BSD 4.3 structure	BSD 4.4/ UNIX 98 compatible structure
<pre>struct sockaddr{ u_short sa_family; char sa_data [14]; }; struct sockaddr_storage{ sa_family_t ss_family; char _ss_pad1[_SS_PAD1SIZE]; char* _ss_align; char _ss_pad2[_SS_PAD2SIZE]; };</pre>	<pre>struct sockaddr { uint8_t sa_len; sa_family_t sa_family; char sa_data[14] }; 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]; };</pre>

Table 3. Address structure

Address structure field	Definition
sa_len	This field contains the length of the address for UNIX 98 specifications. Note: The sa_len field is provided only for BSD 4.4 compatibility. It is not necessary to use this field even for BSD 4.4/UNIX 98 compatibility. The field is ignored on input addresses.
sa_family	This field defines the address family. This value is specified for the address family on the socket() call.
sa_data	This field contains 14 bytes that are reserved to hold the address itself. Note: The sa_data length of 14 bytes is a placeholder for the address. The address can exceed this length. The structure is generic because it does not define the format of the address. The format of the address is defined by the type of transport, which a socket is created for. Each of the transport providers define the exact format for its specific addressing requirements in a similar address structure. The transport is identified by the protocol parameter values on the socket() API.
sockaddr_storage	This field declares storage for any address family address. This structure is large enough and aligned for any protocol-specific structure. It can then be cast as sockaddr structure for use on the APIs. The ss_family field of the sockaddr_storage always aligns with the family field of any protocol-specific structure.

Socket address family

The address family parameter (address_family) on a socket() API determines the format of the address structure to be used on socket APIs.

Address family protocols provide the network transportation of application data from one application to another (or from one process to another within the same system). The application specifies the network transport provider on the protocol parameter of the socket.

AF_INET address family

This address family provides interprocess communication between processes that run on the same system or on different systems.

Addresses for AF_INET sockets are IP addresses and port numbers. You can specify an IP address for an AF_INET socket either as an IP address (such as 130.99.128.1) or in its 32-bit form (X'82638001').

For a socket application that uses the Internet Protocol version 4 (IPv4), the AF_INET address family uses the sockaddr_in address structure. When you use _XOPEN_SOURCE macro, the AF_INET address structure changes to be compatible with BSD 4.4/ UNIX 98 specifications. For the sockaddr_in address structure, these differences are summarized in the table:

Table 4. Differences between BSD 4.3 and BSD 4.4/ UNIX 98 for `sockaddr_in` address structure

BSD 4.3 <code>sockaddr_in</code> address structure	BSD 4.4/ UNIX 98 <code>sockaddr_in</code> address structure
<pre>struct sockaddr_in { short sin_family; u_short sin_port; struct in_addr sin_addr; char sin_zero[8]; };</pre>	<pre>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]; };</pre>

Table 5. `AF_INET` address structure

Address structure field	Definition
<code>sin_len</code>	This field contains the length of the address for UNIX 98 specifications. Note: The <code>sin_len</code> field is provided only for BSD 4.4 compatibility. It is not necessary to use this field even for BSD 4.4/ UNIX 98 compatibility. The field is ignored on input addresses.
<code>sin_family</code>	This field contains the address family, which is always <code>AF_INET</code> when TCP or User Datagram Protocol (UDP) is used.
<code>sin_port</code>	This field contains the port number.
<code>sin_addr</code>	This field contains the IP address.
<code>sin_zero</code>	This field is reserved. Set this field to hexadecimal zeros.

Related reference:

“Using `AF_INET` address family” on page 27

`AF_INET` address family sockets can be either connection-oriented (type `SOCK_STREAM`) or connectionless (type `SOCK_DGRAM`). Connection-oriented `AF_INET` sockets use Transmission Control Protocol (TCP) as the transport protocol. Connectionless `AF_INET` sockets use User Datagram Protocol (UDP) as the transport protocol.

`AF_INET6` address family

This address family provides support for the Internet Protocol version 6 (IPv6). `AF_INET6` address family uses a 128 bit (16 byte) address.

The basic architecture of these addresses includes 64 bits for a network number and another 64 bits for the host number. You can specify `AF_INET6` addresses as `x::x::x::x::x::x`, where the `x`'s are the hexadecimal values of eight 16-bit pieces of the address. For example, a valid address looks like this:
FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

For a socket application that uses TCP, User Datagram Protocol (UDP) or RAW, the `AF_INET6` address family uses the `sockaddr_in6` address structure. This address structure changes if you use `_XOPEN_SOURCE` macro to implement BSD 4.4/ UNIX 98 specifications. For the `sockaddr_in6` address structure, these differences are summarized in this table:

Table 6. Differences between BSD 4.3 and BSD 4.4/ UNIX 98 for `sockaddr_in6` address structure

BSD 4.3 <code>sockaddr_in6</code> address structure	BSD 4.4/ UNIX 98 <code>sockaddr_in6</code> address structure
<pre>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; };</pre>	<pre>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; };</pre>

Table 7. `AF_INET6` address structure

Address structure field	Definition
<code>sin6_len</code>	This field contains the length of the address for UNIX 98 specifications. Note: The <code>sin6_len</code> field is provided only for BSD 4.4 compatibility. It is not necessary to use this field even for BSD 4.4/ UNIX 98 compatibility. The field is ignored on input addresses.
<code>sin6_family</code>	This field specifies the <code>AF_INET6</code> address family.
<code>sin6_port</code>	This field contains the transport layer port.
<code>sin6_flowinfo</code>	This field contains two pieces of information: the traffic class and the flow label. Note: Currently, this field is not supported and should be set to zero to be compatible with later versions.
<code>sin6_addr</code>	This field specifies the IPv6 address.
<code>sin6_scope_id</code>	This field identifies a set of interfaces as appropriate for the scope of the address carried in the <code>sin6_addr</code> field.

AF_UNIX address family

This address family provides interprocess communication on the same system that uses the socket APIs. The address is actually a path name to an entry in the file system.

You can create sockets in the root directory or any open file system but file systems such as QSYS or QDOC. The program must bind an `AF_UNIX`, `SOCK_DGRAM` socket to a name to receive any datagrams back. In addition, the program must explicitly remove the file system object with the `unlink()` API when the socket is closed.

Sockets with the address family `AF_UNIX` use the `sockaddr_un` address structure. This address structure changes if you use `_XOPEN_SOURCE` macro to implement BSD 4.4/ UNIX 98 specifications. For the `sockaddr_un` address structure, these differences are summarized in the table:

Table 8. Differences between BSD 4.3 and BSD 4.4/ UNIX 98 for `sockaddr_un` address structure

BSD 4.3 <code>sockaddr_un</code> address structure	BSD 4.4/ UNIX 98 <code>sockaddr_un</code> address structure
<pre>struct sockaddr_un { short sun_family; char sun_path[126]; };</pre>	<pre>struct sockaddr_un { uint8_t sun_len; sa_family_t sun_family; char sun_path[126]; };</pre>

Table 9. AF_UNIX address structure

Address structure field	Definition
sun_len	This field contains the length of the address for UNIX 98 specifications. Note: The sun_len field is provided only for BSD 4.4 compatibility. It is not necessary to use this field even for BSD 4.4/ UNIX 98 compatibility. The field is ignored on input addresses.
sun_family	This field contains the address family.
sun_path	This field contains the path name to an entry in the file system.

For the AF_UNIX address family, protocol specifications do not apply because protocol standards are not involved. The communications mechanism that the two processes use is specific to the system.

Related reference:

“Using AF_UNIX address family” on page 29

Sockets that use the AF_UNIX or AF_UNIX_CCSID address family can be connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM).

“AF_UNIX_CCSID address family”

The AF_UNIX_CCSID family is compatible with the AF_UNIX address family and has the same limitations.

Related information:

unlink()--Remove Link to File API

AF_UNIX_CCSID address family

The AF_UNIX_CCSID family is compatible with the AF_UNIX address family and has the same limitations.

They both can be either connectionless or connection-oriented, and no external communication functions connect the two processes. The difference is that sockets with the address family AF_UNIX_CCSID use the sockaddr_unc address structure. This address structure is similar to sockaddr_un, but it allows path names in UNICODE or any CCSID by using the Qlg_Path_Name_T format.

However, because an AF_UNIX socket might return the path name from an AF_UNIX_CCSID socket in an AF_UNIX address structure, path size is limited. AF_UNIX supports only 126 characters, so AF_UNIX_CCSID is also limited to 126 characters.

A user cannot exchange AF_UNIX and AF_UNIX_CCSID addresses on a single socket. When AF_UNIX_CCSID is specified on the socket() call, all addresses must be sockaddr_unc on later API calls.

```
struct sockaddr_unc {
    short      sunc_family;
    short      sunc_format;
    char       sunc_zero[12];
    Qlg_Path_Name_T sunc_qlg;
    union {
        char      unix[126];
        wchar_t    wide[126];
        char*      p_unix;
        wchar_t*    p_wide;
    }
    sunc_path;
};
```

Table 10. AF_UNIX_CCSID address structure

Address structure field	Definition
sunc_family	This field contains the address family, which is always AF_UNIX_CCSID.
sunc_format	This field contains two defined values for the format of the path name: <ul style="list-style-type: none"> SO_UNC_DEFAULT indicates a wide path name using the current default CCSID for integrated file system path names. The sunc_qlg field is ignored. SO_UNC_USE_QLG indicates that the sunc_qlg field defines the format and CCSID of the path name.
sunc_zero	This field is reserved. Set this field to hexadecimal zeros.
sunc_qlg	This field specifies the path name format.
sunc_path	This field contains the path name. It is a maximum of 126 characters and can be single byte or double byte. It can be contained within the sunc_path field or allocated separately and pointed to by sunc_path . The format is determined by sunc_format and sunc_qlg .

Related reference:

“Using AF_UNIX_CCSID address family” on page 36

AF_UNIX_CCSID address family sockets have the same specifications as AF_UNIX address family sockets. AF_UNIX_CCSID address family sockets can be connection-oriented or connectionless. They can provide communication on the same system.

“AF_UNIX address family” on page 10

This address family provides interprocess communication on the same system that uses the socket APIs. The address is actually a path name to an entry in the file system.

Related information:

Path name format

Socket type

The second parameter on a socket call determines the socket type. Socket type provides the type identification and characteristics of the connection that are enabled for data transportation from one machine or process to another.

The system supports the following socket types:

Stream (SOCK_STREAM)

This type of socket is connection-oriented. Establish an end-to-end connection by using the bind(), listen(), accept(), and connect() APIs. SOCK_STREAM sends data without errors or duplication, and receives the data in the sending order. SOCK_STREAM builds flow control to avoid data overruns. It does not impose record boundaries on the data. SOCK_STREAM considers the data to be a stream of bytes. In the IBM i implementation, you can use stream sockets over Transmission Control Protocol (TCP), AF_UNIX, and AF_UNIX_CCSID. You can also use stream sockets to communicate with systems outside a secure host (firewall).

Datagram (SOCK_DGRAM)

In Internet Protocol terminology, the basic unit of data transfer is a *datagram*. This is basically a header followed by some data. The datagram socket is connectionless. It establishes no end-to-end connection with the transport provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. You might lose or duplicate data. Datagrams might arrive out of order. The size of

the datagram is limited to the data size that you can send in a single transaction. For some transport providers, each datagram can use a different route through the network. You can issue a `connect()` API on this type of socket. However, on the `connect()` API, you must specify the destination address that the program sends to and receives from. In the IBM i implementation, you can use datagram sockets over User Datagram Protocol (UDP), `AF_UNIX`, and `AF_UNIX_CCSID`.

Raw (SOCK_RAW)

This type of socket allows direct access to lower-layer protocols, such as Internet Protocol (IPv4 or IPv6) and Internet Control Message Protocol (ICMP or ICMP6). `SOCK_RAW` requires more programming expertise because you manage the protocol header information used by the transport provider. At this level, the transport provider can dictate the format of the data and the semantics that are transport-provider specific.

Socket protocols

Socket protocols provide the network transportation of application data from one machine to another (or from one process to another within the same machine).

The application specifies the transport provider on the **protocol** parameter of the `socket()` API.

For the `AF_INET` address family, more than one transport provider is allowed. The protocols of Systems Network Architecture (SNA) and TCP/IP can be active on the same listening socket at the same time. The `ALWANYNET` (Allow ANYNET support) network attribute allows a customer to select whether a transport other than TCP/IP can be used for `AF_INET` socket applications. This network attribute can be either `*YES` or `*NO`. The default value is `*NO`.

For example, if the current status (the default status) is `*NO`, the use of `AF_INET` over an SNA transport is not active. If `AF_INET` sockets are to be used over a TCP/IP transport only, the `ALWANYNET` status should be set to `*NO` to improve CPU utilization.

Note: The `ALWANYNET` network attribute also affects APPC over TCP/IP.

The `AF_INET` and `AF_INET6` sockets over TCP/IP can also specify a `SOCK_RAW` type, which means that the socket communicates directly with the network layer known as Internet Protocol (IP). The TCP or UDP transport providers normally communicate with this layer. When you use `SOCK_RAW` sockets, the application program specifies any protocol between 0 and 255 (except the TCP and UDP protocols). This protocol number then flows in the IP headers when machines are communicating on the network. In fact, the application program is the transport provider, because it must provide for all the transport services that UDP or TCP transports normally provide.

For the `AF_UNIX` and `AF_UNIX_CCSID` address families, a protocol specification is not really meaningful because there are no protocol standards involved. The communications mechanism between two processes on the same machine is specific to the machine.

Related information:

Configuring APPC, APPN, and HPR

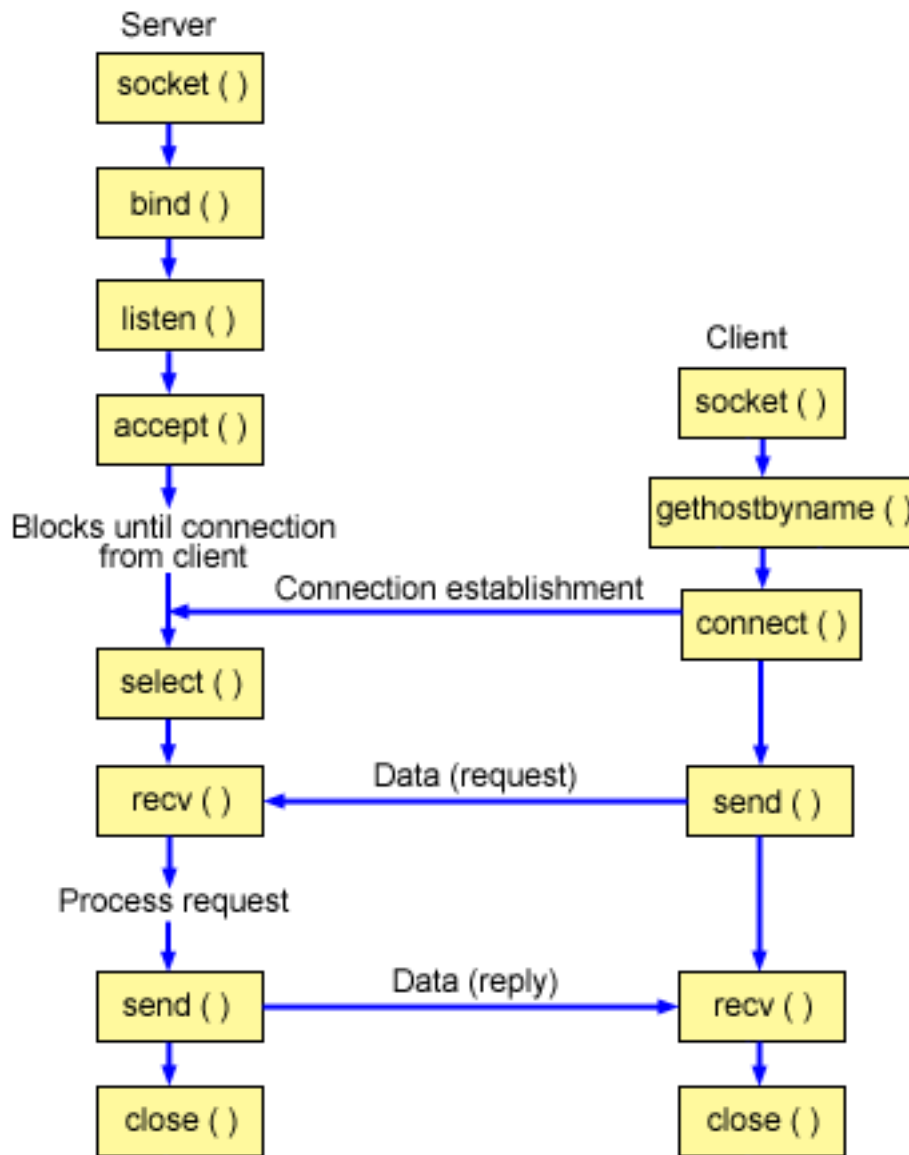
Basic socket design

These examples illustrate the most common types of socket programs that use the most basic design, which can be a basis for more complex socket designs.

Creating a connection-oriented socket

These server and client examples illustrate the socket APIs written for a connection-oriented protocol such as Transmission Control Protocol (TCP).

The following figure illustrates the client/server relationship of the sockets API for a connection-oriented protocol.



Socket flow of events: Connection-oriented server

The following sequence of the socket calls provides a description of the figure. It also describes the relationship between the server and client application in a connection-oriented design. Each set of flows contains links to usage notes on specific APIs.

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the Internet Protocol version 6 address family (`AF_INET6`) with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. The `setsockopt()` API allows the local address to be reused when the server is restarted before the required wait time expires.
3. After the socket descriptor is created, the `bind()` API gets a unique name for the socket. In this example, the user sets the `s6_addr` to zero, which allows connections to be established from any IPv4 or IPv6 client that specifies port 3005.

4. The `listen()` API allows the server to accept incoming client connections. In this example, the backlog is set to 10. This means that the system queues 10 incoming connection requests before the system starts rejecting the incoming requests.
5. The server uses the `accept()` API to accept an incoming connection request. The `accept()` call blocks indefinitely, waiting for the incoming connection to arrive.
6. The `select()` API allows the process to wait for an event to occur and to wake up the process when the event occurs. In this example, the system notifies the process only when data is available to be read. A 30-second timeout is used on this `select()` call.
7. The `recv()` API receives data from the client application. In this example, the client sends 250 bytes of data. Thus, the `SO_RCVLOWAT` socket option can be used, which specifies that `recv()` does not wake up until all 250 bytes of data have arrived.
8. The `send()` API echoes the data back to the client.
9. The `close()` API closes any open socket descriptors.

Socket flow of events: Connection-oriented client

The following sequence of APIs calls describes the relationship between the server and client application in a connection-oriented design.

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the Internet Protocol version 6 address family (`AF_INET6`) with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. In the client example program, if the server string that was passed into the `inet_pton()` API was not a valid IPv6 address string, then it is assumed to be the host name of the server. In that case, use the `getaddrinfo()` API to retrieve the IP address of the server.
3. After the socket descriptor is received, the `connect()` API is used to establish a connection to the server.
4. The `send()` API sends 250 bytes of data to the server.
5. The `recv()` API waits for the server to echo the 250 bytes of data back. In this example, the server responds with the same 250 bytes that was just sent. In the client example, the 250 bytes of the data might arrive in separate packets, so the `recv()` API can be used over and over until all 250 bytes have arrived.
6. The `close()` API closes any open socket descriptors.

Related information:

`listen()`--Invite Incoming Connections Requests API

`bind()`--Set Local Address for Socket API

`accept()`--Wait for Connection Request and Make Connection API

`send()`--Send Data API

`recv()`--Receive Data API

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`setsockopt()`--Set Socket Options API

`select()`--Wait for Events on Multiple Sockets API

`gethostbyname()`--Get Host Information for Host Name API

`connect()`--Establish Connection or Destination Address API

Example: A connection-oriented server

This example shows how a connection-oriented server can be created.

You can use this example to create your own socket server application. A connection-oriented server design is one of the most common models for socket applications. In a connection-oriented design, the server application creates a socket to accept client requests.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This sample program provides a code for a connection-oriented server. */
*****/

/*****
/* Header files needed for this sample program . */
*****/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/poll.h>

/*****
/* Constants used by this program */
*****/
#define SERVER_PORT 12345
#define BUFFER_LENGTH 250
#define FALSE 0

void main()
{
    /*****
    /* Variable and structure definitions. */
    *****/
    int sd=-1, sd2=-1;
    int rc, length, on=1;
    char buffer[BUFFER_LENGTH];
    struct pollfd fds;
    nfds_t nfds = 1;
    int timeout;
    struct sockaddr_in6 serveraddr;

    /*****
    /* A do/while(FALSE) loop is used to make error cleanup easier. The */
    /* close() of each of the socket descriptors is only done once at the */
    /* very end of the program. */
    *****/
    do
    {
        /*****
        /* The socket() function returns a socket descriptor, representing */
        /* an endpoint. The statement also identifies that the INET6 */
        /* (Internet Protocol version 6) address family with the TCP */
        /* transport (SOCK_STREAM) will be used for this socket. */
        *****/
        sd = socket(AF_INET6, SOCK_STREAM, 0);
        if (sd < 0)
        {
            perror("socket() failed");
            break;
        }

        /*****
        /* The setsockopt() function is used to allow the local address to */
        /* be reused when the server is restarted before the required wait */
        /* time expires. */
        *****/
        rc = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
    }
}
```

```

if (rc < 0)
{
    perror("setsockopt(SO_REUSEADDR) failed");
    break;
}

/*****
/* After the socket descriptor is created, a bind() function gets a */
/* unique name for the socket. In this example, the user sets the */
/* s6_addr to zero, which allows connections to be established from */
/* any client that specifies port 12345. */
*****/
memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin6_family = AF_INET6;
serveraddr.sin6_port = htons(SERVER_PORT);
memcpy(&serveraddr.sin6_addr, &in6addr_any, sizeof(in6addr_any));

rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
if (rc < 0)
{
    perror("bind() failed");
    break;
}

/*****
/* The listen() function allows the server to accept incoming */
/* client connections. In this example, the backlog is set to 10. */
/* This means that the system will queue 10 incoming connection */
/* requests before the system starts rejecting the incoming */
/* requests. */
*****/
rc = listen(sd, 10);
if (rc < 0)
{
    perror("listen() failed");
    break;
}

printf("Ready for client connect().\n");

/*****
/* The server uses the accept() function to accept an incoming */
/* connection request. The accept() call will block indefinitely */
/* waiting for the incoming connection to arrive. */
*****/
sd2 = accept(sd, NULL, NULL);
if (sd2 < 0)
{
    perror("accept() failed");
    break;
}

/*****
/* The poll() function allows the process to wait for an event to */
/* occur and to wake up the process when the event occurs. In this */
/* example, the system notifies the process only when data is */
/* available to read. A 30 second timeout is used on this poll */
/* call. */
*****/
timeout = 30000;

memset(&fds, 0, sizeof(fds));
fds.fd = sd2;
fds.events = POLLIN;
fds.revents = 0;

rc = poll(&fds, nfd, timeout);

```

```

if (rc < 0)
{
    perror("poll() failed");
    break;
}

if (rc == 0)
{
    printf("poll() timed out.\n");
    break;
}

/*****
/* In this example we know that the client will send 250 bytes of
/* data over. Knowing this, we can use the SO_RCVLOWAT socket
/* option and specify that we don't want our recv() to wake up until
/* all 250 bytes of data have arrived.
*****/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
               (char *)&length, sizeof(length));

if (rc < 0)
{
    perror("setsockopt(SO_RCVLOWAT) failed");
    break;
}

/*****
/* Receive that 250 bytes data from the client
*****/
rc = recv(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("recv() failed");
    break;
}

printf("%d bytes of data were received\n", rc);
if (rc == 0 ||
    rc < sizeof(buffer))
{
    printf("The client closed the connection before all of the\n");
    printf("data was sent\n");
    break;
}

/*****
/* Echo the data back to the client
*****/
rc = send(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("send() failed");
    break;
}

/*****
/* Program complete
*****/

} while (FALSE);

/*****
/* Close down any open socket descriptors
*****/
if (sd != -1)

```



```

        close(sd);
    if (sd2 != -1)
        close(sd2);
}

```

Related reference:

“Using AF_INET address family” on page 27

AF_INET address family sockets can be either connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM). Connection-oriented AF_INET sockets use Transmission Control Protocol (TCP) as the transport protocol. Connectionless AF_INET sockets use User Datagram Protocol (UDP) as the transport protocol.

“Example: A connection-oriented client”

This example shows how to create a socket client program to connect to a connection-oriented server in a connection-oriented design.

Example: A connection-oriented client

This example shows how to create a socket client program to connect to a connection-oriented server in a connection-oriented design.

The client of the service (the client program) must request the service of the server program. You can use this example to write your own client application.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This sample program provides a code for a connection-oriented client. */
*****/

/*****
/* Header files needed for this sample program */
*****/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*****
/* Constants used by this program */
*****/
#define SERVER_PORT      3005
#define BUFFER_LENGTH    250
#define FALSE            0
#define SERVER_NAME      "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or */
/* set the server name in the #define */
/* SERVER_NAME. */
void main(int argc, char *argv[])
{
    /*****
    /* Variable and structure definitions. */
    *****/
    int    sd=-1, rc, bytesReceived;
    char    buffer[BUFFER_LENGTH];
    char    server[NETDB_MAX_HOST_NAME_LENGTH];
    struct sockaddr_in6 serveraddr;
    struct addrinfo hints, *res;

    /*****

```

```

/* A do/while(FALSE) loop is used to make error cleanup easier. The */
/* close() of the socket descriptor is only done once at the very end */
/* of the program. */
/*****/
do
{
    /*****/
    /* The socket() function returns a socket descriptor, representing */
    /* an endpoint. The statement also identifies that the INET6 */
    /* (Internet Protocol version 6) address family with the TCP */
    /* transport (SOCK_STREAM) will be used for this socket. */
    /*****/
    sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (sd < 0)
    {
        perror("socket() failed");
        break;
    }

    /*****/
    /* If an argument was passed in, use this as the server, otherwise */
    /* use the #define that is located at the top of this program. */
    /*****/
    if (argc > 1)
        strcpy(server, argv[1]);
    else
        strcpy(server, SERVER_NAME);

    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_port = htons(SERVER_PORT);
    rc = inet_pton(AF_INET6, server, &serveraddr.sin6_addr.s6_addr);

    if (rc != 1)
    {
        /*****/
        /* The server string that was passed into the inet_pton() */
        /* function was not a hexadecimal colon IP address. It must */
        /* therefore be the hostname of the server. Use the */
        /* getaddrinfo() function to retrieve the IP address of the */
        /* server. */
        /*****/
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET6;
        hints.ai_flags = AI_V4MAPPED;
        rc = getaddrinfo(server, NULL, &hints, &res);
        if (rc != 0)
        {
            printf("Host not found! (%s)\n", server);
            perror("getaddrinfo() failed\n");
            break;
        }

        memcpy(&serveraddr.sin6_addr,
            (&(struct sockaddr_in6 *) (res->ai_addr))->sin6_addr,
            sizeof(serveraddr.sin6_addr));

        freeaddrinfo(res);
    }

    /*****/
    /* Use the connect() function to establish a connection to the */
    /* server. */
    /*****/
    rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
    if (rc < 0)
    {

```

```

        perror("connect() failed");
        break;
    }

    /******
    /* Send 250 bytes of a's to the server */
    /******
    memset(buffer, 'a', sizeof(buffer));
    rc = send(sd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
        perror("send() failed");
        break;
    }

    /******
    /* In this example we know that the server is going to respond with */
    /* the same 250 bytes that we just sent. Since we know that 250 */
    /* bytes are going to be sent back to us, we can use the */
    /* SO_RCVLOWAT socket option and then issue a single recv() and */
    /* retrieve all of the data. */
    /*
    /* The use of SO_RCVLOWAT is already illustrated in the server */
    /* side of this example, so we will do something different here. */
    /* The 250 bytes of the data may arrive in separate packets, */
    /* therefore we will issue recv() over and over again until all */
    /* 250 bytes have arrived. */
    /******
    bytesReceived = 0;
    while (bytesReceived < BUFFER_LENGTH)
    {
        rc = recv(sd, & buffer[bytesReceived],
                  BUFFER_LENGTH - bytesReceived, 0);
        if (rc < 0)
        {
            perror("recv() failed");
            break;
        }
        else if (rc == 0)
        {
            printf("The server closed the connection\n");
            break;
        }

        /******
        /* Increment the number of bytes that have been received so far */
        /******
        bytesReceived += rc;
    }

    } while (FALSE);

    /******
    /* Close down any open socket descriptors */
    /******
    if (sd != -1)
        close(sd);
}

```

Related reference:

“Using AF_INET address family” on page 27

AF_INET address family sockets can be either connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM). Connection-oriented AF_INET sockets use Transmission Control Protocol (TCP) as the transport protocol. Connectionless AF_INET sockets use User Datagram Protocol (UDP) as the transport protocol.

“Example: A connection-oriented server” on page 15

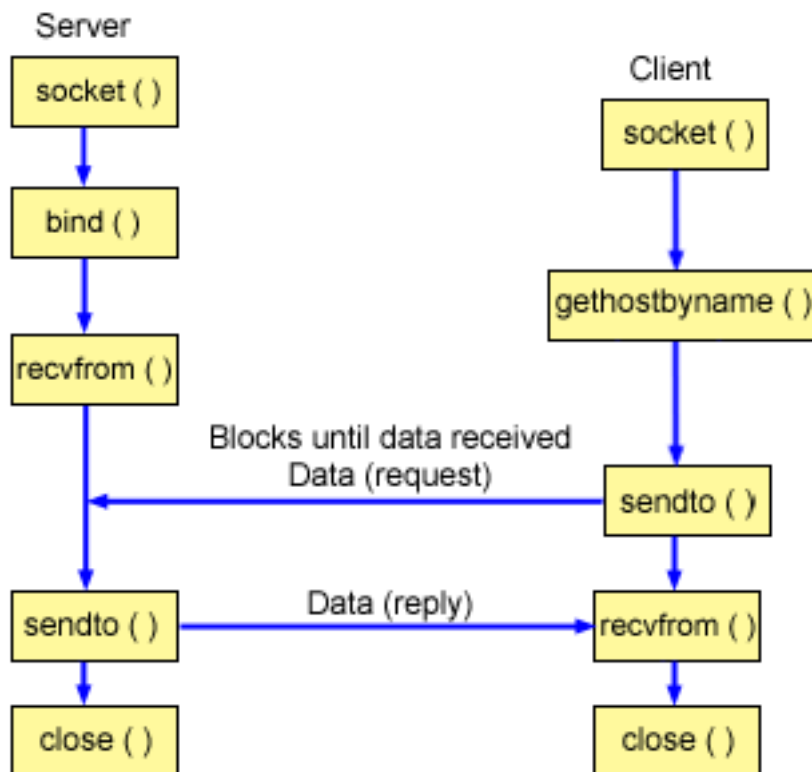
This example shows how a connection-oriented server can be created.

Creating a connectionless socket

Connectionless sockets do not establish a connection over which data is transferred. Instead, the server application specifies its name where a client can send requests.

Connectionless sockets use User Datagram Protocol (UDP) instead of TCP/IP.

The following figure illustrates the client/server relationship of the socket APIs used in the examples for a connectionless socket design.



Socket flow of events: Connectionless server

The following sequence of the socket calls provides a description of the figure and the following example programs. It also describes the relationship between the server and client application in a connectionless design. Each set of flows contains links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The first example of a connectionless server uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the Internet Protocol version 6 address family (AF_INET6) with the UDP transport (SOCK_DGRAM) is used for this socket.
2. After the socket descriptor is created, a `bind()` API gets a unique name for the socket. In this example, the user sets the `s6_addr` to zero, which means that the UDP port of 3555 is bound to all IPv4 and IPv6 addresses on the system.
3. The server uses the `recvfrom()` API to receive that data. The `recvfrom()` API waits indefinitely for data to arrive.

4. The sendto() API echoes the data back to the client.
5. The close() API ends any open socket descriptors.

Socket flow of events: Connectionless client

The second example of a connectionless client uses the following sequence of API calls.

1. The socket() API returns a socket descriptor, which represents an endpoint. The statement also identifies that the Internet Protocol version 6 address family (AF_INET6) with the UDP transport (SOCK_DGRAM) is used for this socket.
2. In the client example program, if the server string that was passed into the inet_pton() API was not a valid IPv6 address string, then it is assumed to be the host name of the server. In that case, use the getaddrinfo() API to retrieve the IP address of the server.
3. Use the sendto() API to send the data to the server.
4. Use the recvfrom() API to receive the data from the server.
5. The close() API ends any open socket descriptors.

Related information:

close()--Close File or Socket Descriptor API

socket()--Create Socket API

bind()--Set Local Address for Socket API

recvfrom()--Receive Data API

sendto()--Send Data API

gethostbyname()--Get Host Information for Host Name API

Example: A connectionless server

This example illustrates how to create a connectionless socket server program by using User Datagram Protocol (UDP).

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This sample program provides a code for a connectionless server.      */
*****/

/*****
/* Header files needed for this sample program                          */
*****/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*****
/* Constants used by this program                                        */
*****/
#define SERVER_PORT    3555
#define BUFFER_LENGTH  100
#define FALSE          0

void main()
{
    /*****
    /* Variable and structure definitions.                                */
    *****/
    int    sd=-1, rc;
    char   buffer[BUFFER_LENGTH];
    struct sockaddr_in6 serveraddr;

```

```

struct sockaddr_in6 clientaddr;
int    clientaddrlen = sizeof(clientaddr);

/*****
/* A do/while(FALSE) loop is used to make error cleanup easier. The
/* close() of each of the socket descriptors is only done once at the
/* very end of the program.
*****/
do
{
    /*****/
    /* The socket() function returns a socket descriptor, representing
    /* an endpoint. The statement also identifies that the INET6
    /* (Internet Protocol version 6) address family with the UDP
    /* transport (SOCK_DGRAM) will be used for this socket.
    *****/
    sd = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sd < 0)
    {
        perror("socket() failed");
        break;
    }

    /*****/
    /* After the socket descriptor is created, a bind() function gets a
    /* unique name for the socket. In this example, the user sets the
    /* s_addr to zero, which means that the UDP port of 3555 will be
    /* bound to all IP addresses on the system.
    *****/
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_port = htons(SERVER_PORT);
    memcpy(&serveraddr.sin6_addr, &in6addr_any, sizeof(in6addr_any));

    rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
    if (rc < 0)
    {
        perror("bind() failed");
        break;
    }

    /*****/
    /* The server uses the recvfrom() function to receive that data.
    /* The recvfrom() function waits indefinitely for data to arrive.
    *****/
    rc = recvfrom(sd, buffer, sizeof(buffer), 0,
                  (struct sockaddr *)&clientaddr,
                  &clientaddrlen);

    if (rc < 0)
    {
        perror("recvfrom() failed");
        break;
    }

    printf("server received the following: <%s>\n", buffer);
    inet_ntop(AF_INET6, &clientaddr.sin6_addr.s6_addr,
              buffer, sizeof(buffer));
    printf("from port %d and address %s\n",
          ntohs(clientaddr.sin6_port),
          buffer);

    /*****/
    /* Echo the data back to the client
    *****/
    rc = sendto(sd, buffer, sizeof(buffer), 0,
                (struct sockaddr *)&clientaddr,
                sizeof(clientaddr));

```

```

    if (rc < 0)
    {
        perror("sendto() failed");
        break;
    }

    /*****
    /* Program complete
    *****/

} while (FALSE);

/*****
/* Close down any open socket descriptors
*****/
if (sd != -1)
    close(sd);
}

```

Example: A connectionless client

This example shows how to use User Datagram Protocol (UDP) to connect a connectionless socket client program to a server.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This sample program provides a code for a connectionless client.
*****/

/*****
/* Header files needed for this sample program
*****/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*****
/* Constants used by this program
*****/
#define SERVER_PORT    3555
#define BUFFER_LENGTH  100
#define FALSE         0
#define SERVER_NAME    "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or */
/* set the server name in the #define */
/* SERVER_NAME */
void main(int argc, char *argv[])
{
    /*****
    /* Variable and structure definitions.
    *****/
    int    sd, rc;
    char    server[NETDB_MAX_HOST_NAME_LENGTH];
    char    buffer[BUFFER_LENGTH];
    struct sockaddr_in6 serveraddr;
    int    serveraddrlen = sizeof(serveraddr);
    struct addrinfo hints, *res;

    /*****

```

```

/* A do/while(FALSE) loop is used to make error cleanup easier. The */
/* close() of the socket descriptor is only done once at the very end */
/* of the program. */
/*****/
do
{
    /*****/
    /* The socket() function returns a socket descriptor, representing */
    /* an endpoint. The statement also identifies that the INET6 */
    /* (Internet Protocol) address family with the UDP transport */
    /* (SOCK_DGRAM) will be used for this socket. */
    /*****/
    sd = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sd < 0)
    {
        perror("socket() failed");
        break;
    }

    /*****/
    /* If an argument was passed in, use this as the server, otherwise */
    /* use the #define that is located at the top of this program. */
    /*****/
    if (argc > 1)
        strcpy(server, argv[1]);
    else
        strcpy(server, SERVER_NAME);

    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_port = htons(SERVER_PORT);
    rc = inet_pton(AF_INET6, server, &serveraddr.sin6_addr.s6_addr);
    if (rc != 1)
    {
        /*****/
        /* The server string that was passed into the inet_pton() */
        /* function was not a hexadecimal colon IP address. It must */
        /* therefore be the hostname of the server. Use the */
        /* getaddrinfo() function to retrieve the IP address of the */
        /* server. */
        /*****/
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET6;
        hints.ai_flags = AI_V4MAPPED;
        rc = getaddrinfo(server, NULL, &hints, &res);
        if (rc != 0)
        {
            printf("Host not found! (%s)", server);
            break;
        }

        memcpy(&serveraddr.sin6_addr,
            (&(struct sockaddr_in6 *) (res->ai_addr))->sin6_addr,
            sizeof(serveraddr.sin6_addr));

        freeaddrinfo(res);
    }

    /*****/
    /* Initialize the data block that is going to be sent to the server */
    /*****/
    memset(buffer, 0, sizeof(buffer));
    strcpy(buffer, "A CLIENT REQUEST");

    /*****/
    /* Use the sendto() function to send the data to the server. */
    /*****/

```



```

rc = sendto(sd, buffer, sizeof(buffer), 0,
            (struct sockaddr *)&serveraddr,
            sizeof(serveraddr));
if (rc < 0)
{
    perror("sendto() failed");
    break;
}

/*****
/* Use the recvfrom() function to receive the data back from the
/* server.
*****/
rc = recvfrom(sd, buffer, sizeof(buffer), 0,
              (struct sockaddr *)&serveraddr,
              & serveraddrlen);
if (rc < 0)
{
    perror("recvfrom() failed");
    break;
}

printf("client received the following: <%s>\n", buffer);
inet_ntop(AF_INET6, &serveraddr.sin6_addr.s6_addr,
          buffer, sizeof(buffer));
printf("from port %d, from address %s\n",
       ntohs(serveraddr.sin6_port), buffer);

/*****
/* Program complete
*****/

} while (FALSE);

/*****
/* Close down any open socket descriptors
*****/
if (sd != -1)
    close(sd);
}

```

Designing applications with address families

These scenarios illustrate how to design applications with each of the socket address families, such as AF_INET address family, AF_INET6 address family, AF_UNIX address family, and AF_UNIX_CCSID address family.

Using AF_INET address family

AF_INET address family sockets can be either connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM). Connection-oriented AF_INET sockets use Transmission Control Protocol (TCP) as the transport protocol. Connectionless AF_INET sockets use User Datagram Protocol (UDP) as the transport protocol.

When you create an AF_INET domain socket, you specify AF_INET for the address family in the socket program. AF_INET sockets can also use a type of SOCK_RAW. If this type is set, the application connects directly to the IP layer and does not use either the TCP or UDP transport.

Related reference:

“AF_INET address family” on page 8

This address family provides interprocess communication between processes that run on the same system or on different systems.

“Prerequisites for socket programming” on page 3

Before writing socket applications, you must complete these steps to meet the requirements for compiler, AF_INET and AF_INET6 address families, Secure Sockets Layer (SSL) APIs, and Global Security Kit (GSKit) APIs.

“Example: A connection-oriented server” on page 15

This example shows how a connection-oriented server can be created.

“Example: A connection-oriented client” on page 19

This example shows how to create a socket client program to connect to a connection-oriented server in a connection-oriented design.

Using AF_INET6 address family

AF_INET6 sockets provide support for Internet Protocol version 6 (IPv6) 128 bit (16 byte) address structures. Programmers can write applications using the AF_INET6 address family to accept client requests from either IPv4 or IPv6 nodes, or from IPv6 nodes only.

Like AF_INET sockets, AF_INET6 sockets can be either connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM). Connection-oriented AF_INET6 sockets use TCP as the transport protocol. Connectionless AF_INET6 sockets use User Datagram Protocol (UDP) as the transport protocol. When you create an AF_INET6 domain socket, you specify AF_INET6 for the address family in the socket program. AF_INET6 sockets can also use a type of SOCK_RAW. If this type is set, the application connects directly to the IP layer and does not use either the TCP or UDP transport.

IPv6 applications compatibility with IPv4 applications

Socket applications written with AF_INET6 address family allow Internet Protocol version 6 (IPv6) applications to work with Internet Protocol version 4 (IPv4) applications (those applications that use AF_INET address family). This feature allows socket programmers to use an IPv4-mapped IPv6 address format. This address format represents the IPv4 address of an IPv4 node to be represented as an IPv6 address. The IPv4 address is encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold the fixed prefix 0:0:0:0:FFFF. For example, an IPv4-mapped address can look like this:

```
::FFFF:192.1.1.1
```

These addresses can be generated automatically by the getaddrinfo() API, when the specified host has only IPv4 addresses.

You can create applications that use AF_INET6 sockets to open TCP connections to IPv4 nodes. To accomplish this task, you can encode the destination's IPv4 address as an IPv4-mapped IPv6 address and pass that address within a sockaddr_in6 structure in the connect() or sendto() call. When applications use AF_INET6 sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system returns the peer's address to the application in the accept(), recvfrom(), or getpeername() calls using a sockaddr_in6 structure encoded this way.

While the bind() API allows applications to select the source IP address of UDP packets and TCP connections, applications often want the system to select the source address for them. Applications use in6addr_any similarly to the way they use the INADDR_ANY macro in IPv4 for this purpose. An additional feature of binding in this way is that it allows an AF_INET6 socket to communicate with both IPv4 and IPv6 nodes. For example, an application issuing an accept() on a listening socket bound to in6addr_any accepts connections from either IPv4 or IPv6 nodes. This behavior can be modified through the use of the IPPROTO_IPV6 level socket option IPV6_V6ONLY. Few applications need to know which type of node with which they are interoperating. However, for those applications that do need to know, the IN6_IS_ADDR_V4MAPPED() macro defined in <netinet/in.h> is provided.

Related reference:

“Prerequisites for socket programming” on page 3

Before writing socket applications, you must complete these steps to meet the requirements for compiler, AF_INET and AF_INET6 address families, Secure Sockets Layer (SSL) APIs, and Global Security Kit

(GSKit) APIs.

“Socket scenario: Creating an application to accept IPv4 and IPv6 clients” on page 79

This example shows a typical situation in which you might want to use the AF_INET6 address family.

Related information:

Comparison of IPv4 and IPv6

recvfrom()--Receive Data API

accept()--Wait for Connection Request and Make Connection API

getpeername()--Retrieve Destination Address of Socket API

sendto()--Send Data API

connect()--Establish Connection or Destination Address API

bind()--Set Local Address for Socket API

gethostbyname()--Get Host Information for Host Name API

getaddrinfo()--Get Address Information API

gethostbyaddr()--Get Host Information for IP Address API

getnameinfo()--Get Name Information for Socket Address API

Using AF_UNIX address family

Sockets that use the AF_UNIX or AF_UNIX_CCSID address family can be connection-oriented (type SOCK_STREAM) or connectionless (type SOCK_DGRAM).

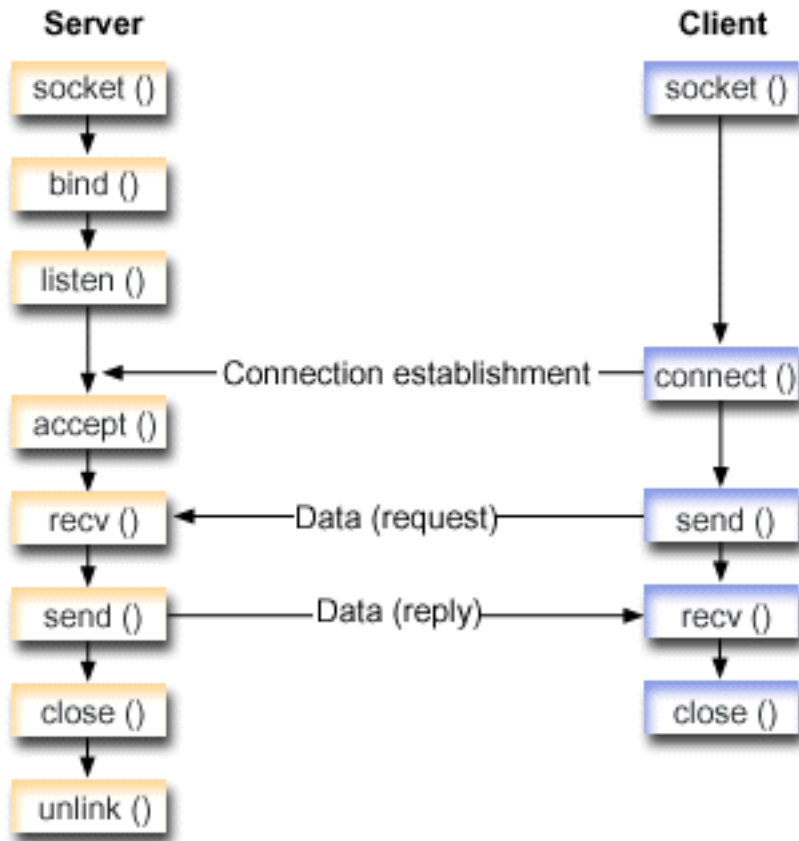
Both types are reliable because there are no external communication functions connecting the two processes.

UNIX domain datagram sockets act differently from UDP datagram sockets. With UDP datagram sockets, the client program does not need to call the bind() API because the system assigns an unused port number automatically. The server can then send a datagram back to that port number. However, with UNIX domain datagram sockets, the system does not automatically assign a path name for the client. Thus, all client programs using UNIX domain datagrams must call the bind() API. The exact path name specified on the client's bind() is what is passed to the server. Thus, if the client specifies a relative path name (that is, a path name that is not fully qualified by starting with /), the server cannot send the client a datagram unless it is running with the same current directory.

An example path name that an application might use for this address family is /tmp/myserver or servers/thatserver. With servers/thatserver, you have a path name that is not fully qualified (no / was specified). This means that the location of the entry in the file system hierarchy should be determined relative to the current working directory.

Note: Path names in the file system are NLS-enabled.

The following figure illustrates the client/server relationship of the AF_UNIX address family.



Socket flow of events: Server application that uses AF_UNIX address family

The first example uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies the UNIX address family with the stream transport (`SOCK_STREAM`) being used for this socket. You can also use the `socketpair()` API to initialize a UNIX socket.
`AF_UNIX` or `AF_UNIX_CCSID` are the only address families to support the `socketpair()` API. The `socketpair()` API returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is created, the `bind()` API gets a unique name for the socket.
The name space for UNIX domain sockets consists of path names. When a sockets program calls the `bind()` API, an entry is created in the file system directory. If the path name already exists, the `bind()` fails. Thus, a UNIX domain socket program should always call an `unlink()` API to remove the directory entry when it ends.
3. The `listen()` allows the server to accept incoming client connections. In this example, the backlog is set to 10. This means that the system queues 10 incoming connection requests before the system starts rejecting the incoming requests.
4. The `recv()` API receives data from the client application. In this example, the client sends 250 bytes of data over. Thus, the `SO_RCVLOWAT` socket option can be used, which specifies that `recv()` is not required to wake up until all 250 bytes of data have arrived.
5. The `send()` API echoes the data back to the client.
6. The `close()` API closes any open socket descriptors.
7. The `unlink()` API removes the UNIX path name from the file system.

Socket flow of events: Client application that uses AF_UNIX address family

The second example uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies the UNIX address family with the stream transport (`SOCK_STREAM`) being used for this socket. You can also use the `socketpair()` API to initialize a UNIX socket.
AF_UNIX or AF_UNIX_CCSID are the only address families to support the `socketpair()` API. The `socketpair()` API returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is received, the `connect()` API is used to establish a connection to the server.
3. The `send()` API sends 250 bytes of data that are specified in the server application with the `SO_RCVLOWAT` socket option.
4. The `recv()` API loops until all 250 bytes of the data have arrived.
5. The `close()` API closes any open socket descriptors.

Related reference:

“AF_UNIX address family” on page 10

This address family provides interprocess communication on the same system that uses the socket APIs. The address is actually a path name to an entry in the file system.

“Prerequisites for socket programming” on page 3

Before writing socket applications, you must complete these steps to meet the requirements for compiler, AF_INET and AF_INET6 address families, Secure Sockets Layer (SSL) APIs, and Global Security Kit (GSKit) APIs.

“Using AF_UNIX_CCSID address family” on page 36

AF_UNIX_CCSID address family sockets have the same specifications as AF_UNIX address family sockets. AF_UNIX_CCSID address family sockets can be connection-oriented or connectionless. They can provide communication on the same system.

Related information:

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`bind()`--Set Local Address for Socket API

`unlink()`--Remove Link to File API

`listen()`--Invite Incoming Connections Requests API

`send()`--Send Data API

`recv()`--Receive Data API

`socketpair()`--Create a Pair of Sockets API

`connect()`--Establish Connection or Destination Address API

Example: Server application that uses AF_UNIX address family:

This example illustrates a sample server program that uses the AF_UNIX address family. The AF_UNIX address family uses many of the same socket calls as other address families, except that it uses the path name structure to identify the server application.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/*
*****
/* This example program provides code for a server application that uses
/* AF_UNIX address family
*****
Header files needed for this sample program
*/
```

```

/*****/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*****/
/* Constants used by this program */
/*****/
#define SERVER_PATH    "/tmp/server"
#define BUFFER_LENGTH  250
#define FALSE          0

void main()
{
    /*****/
    /* Variable and structure definitions. */
    /*****/
    int    sd=-1, sd2=-1;
    int    rc, length;
    char    buffer[BUFFER_LENGTH];
    struct sockaddr_un serveraddr;

    /*****/
    /* A do/while(FALSE) loop is used to make error cleanup easier. The */
    /* close() of each of the socket descriptors is only done once at the */
    /* very end of the program. */
    /*****/
    do
    {
        /*****/
        /* The socket() function returns a socket descriptor, which represents */
        /* an endpoint. The statement also identifies that the UNIX */
        /* address family with the stream transport (SOCK_STREAM) will be */
        /* used for this socket. */
        /*****/
        sd = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sd < 0)
        {
            perror("socket() failed");
            break;
        }

        /*****/
        /* After the socket descriptor is created, a bind() function gets a */
        /* unique name for the socket. */
        /*****/
        memset(&serveraddr, 0, sizeof(serveraddr));
        serveraddr.sun_family = AF_UNIX;
        strcpy(serveraddr.sun_path, SERVER_PATH);

        rc = bind(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
        if (rc < 0)
        {
            perror("bind() failed");
            break;
        }

        /*****/
        /* The listen() function allows the server to accept incoming */
        /* client connections. In this example, the backlog is set to 10. */
        /* This means that the system will queue 10 incoming connection */
        /* requests before the system starts rejecting the incoming */
        /* requests. */
        /*****/
        rc = listen(sd, 10);

```

```

if (rc < 0)
{
    perror("listen() failed");
    break;
}

printf("Ready for client connect().\n");

/*****
/* The server uses the accept() function to accept an incoming
/* connection request. The accept() call will block indefinitely
/* waiting for the incoming connection to arrive.
*****/
sd2 = accept(sd, NULL, NULL);
if (sd2 < 0)
{
    perror("accept() failed");
    break;
}

/*****
/* In this example we know that the client will send 250 bytes of
/* data over. Knowing this, we can use the SO_RCVLOWAT socket
/* option and specify that we don't want our recv() to wake up
/* until all 250 bytes of data have arrived.
*****/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                (char *)&length, sizeof(length));

if (rc < 0)
{
    perror("setsockopt(SO_RCVLOWAT) failed");
    break;
}

/*****
/* Receive that 250 bytes data from the client
*****/
rc = recv(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("recv() failed");
    break;
}

printf("%d bytes of data were received\n", rc);
if (rc == 0 ||
    rc < sizeof(buffer))
{
    printf("The client closed the connection before all of the\n");
    printf("data was sent\n");
    break;
}

/*****
/* Echo the data back to the client
*****/
rc = send(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("send() failed");
    break;
}

/*****
/* Program complete
*****/

} while (FALSE);

```

```

/*****
/* Close down any open socket descriptors */
/*****
if (sd != -1)
    close(sd);

if (sd2 != -1)
    close(sd2);

/*****
/* Remove the UNIX path name from the file system */
/*****
unlink(SERVER_PATH);
}

```

Example: Client application that uses AF_UNIX address family:

This example shows a sample application that uses the AF_UNIX address family to create a client connection to a server.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This sample program provides code for a client application that uses */
/* AF_UNIX address family */
/*****
/*****
/* Header files needed for this sample program */
/*****
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*****
/* Constants used by this program */
/*****
#define SERVER_PATH    "/tmp/server"
#define BUFFER_LENGTH  250
#define FALSE          0

/* Pass in 1 parameter which is either the */
/* path name of the server as a UNICODE */
/* string, or set the server path in the */
/* #define SERVER_PATH which is a CCSID */
/* 500 string. */
void main(int argc, char *argv[])
{
    /*****
    /* Variable and structure definitions. */
    /*****
    int    sd=-1, rc, bytesReceived;
    char    buffer[BUFFER_LENGTH];
    struct sockaddr_un serveraddr;

    /*****
    /* A do/while(FALSE) loop is used to make error cleanup easier. The */
    /* close() of the socket descriptor is only done once at the very end */
    /* of the program. */
    /*****
    do
    {
        /*****

```



```

/* The socket() function returns a socket descriptor, which represents */
/* an endpoint. The statement also identifies that the UNIX */
/* address family with the stream transport (SOCK_STREAM) will be */
/* used for this socket. */
/*****
sd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sd < 0)
{
    perror("socket() failed");
    break;
}

/*****
/* If an argument was passed in, use this as the server, otherwise */
/* use the #define that is located at the top of this program. */
/*****
memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sun_family = AF_UNIX;
if (argc > 1)
    strcpy(serveraddr.sun_path, argv[1]);
else
    strcpy(serveraddr.sun_path, SERVER_PATH);

/*****
/* Use the connect() function to establish a connection to the */
/* server. */
/*****
rc = connect(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
if (rc < 0)
{
    perror("connect() failed");
    break;
}

/*****
/* Send 250 bytes of a's to the server */
/*****
memset(buffer, 'a', sizeof(buffer));
rc = send(sd, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("send() failed");
    break;
}

/*****
/* In this example we know that the server is going to respond with */
/* the same 250 bytes that we just sent. Since we know that 250 */
/* bytes are going to be sent back to us, we can use the */
/* SO_RCVLOWAT socket option and then issue a single recv() and */
/* retrieve all of the data. */
/*
/* The use of SO_RCVLOWAT is already illustrated in the server */
/* side of this example, so we will do something different here. */
/* The 250 bytes of the data may arrive in separate packets, */
/* therefore we will issue recv() over and over again until all */
/* 250 bytes have arrived. */
/*****
bytesReceived = 0;
while (bytesReceived < BUFFER_LENGTH)
{
    rc = recv(sd, & buffer[bytesReceived],
              BUFFER_LENGTH - bytesReceived, 0);
    if (rc < 0)
    {
        perror("recv() failed");
        break;
    }
}

```

```

    }
    else if (rc == 0)
    {
        printf("The server closed the connection\n");
        break;
    }

    /*****
    /* Increment the number of bytes that have been received so far */
    *****/
    bytesReceived += rc;
}

} while (FALSE);

/*****
/* Close down any open socket descriptors */
*****/
if (sd != -1)
    close(sd);
}

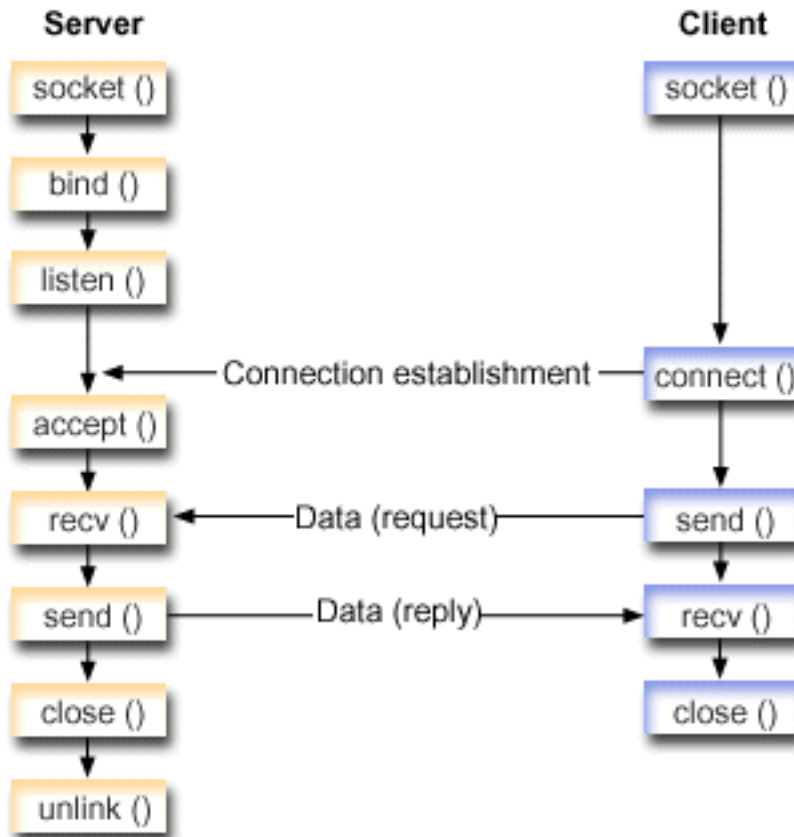
```

Using AF_UNIX_CCSID address family

AF_UNIX_CCSID address family sockets have the same specifications as AF_UNIX address family sockets. AF_UNIX_CCSID address family sockets can be connection-oriented or connectionless. They can provide communication on the same system.

Before working with an AF_UNIX_CCSID socket application, you must be familiar with the Qlg_Path_Name_T structure to determine the output format.

When working with an output address structure, such as one returned from accept(), getsockname(), getpeername(), recvfrom(), and recvmsg(), the application must examine the socket address structure (sockaddr_unc) to determine its format. The **sunc_format** and **sunc_qlg** fields determine the output format of the path name. But sockets do not necessarily use the same values on output as the application used on input addresses.



Socket flow of events: Server application that uses AF_UNIX_CCSID address family

The first example uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the `UNIX_CCSID` address family with the stream transport (`SOCK_STREAM`) is used for this socket. You can also use the `socketpair()` API to initialize a UNIX socket.
`AF_UNIX` or `AF_UNIX_CCSID` are the only address families to support the `socketpair()` API. The `socketpair()` API returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is created, the `bind()` API gets a unique name for the socket.
The name space for UNIX domain sockets consists of path names. When a sockets program calls the `bind()` API, an entry is created in the file system directory. If the path name already exists, the `bind()` fails. Thus, a UNIX domain socket program should always call an `unlink()` API to remove the directory entry when it ends.
3. The `listen()` allows the server to accept incoming client connections. In this example, the backlog is set to 10. This means that the system queues 10 incoming connection requests before the system starts rejecting the incoming requests.
4. The server uses the `accept()` API to accept an incoming connection request. The `accept()` call blocks indefinitely, waiting for the incoming connection to arrive.
5. The `recv()` API receives data from the client application. In this example, the client sends 250 bytes of data over. Thus, the `SO_RCVLOWAT` socket option can be used, which specifies that `recv()` is not required to wake up until all 250 bytes of data have arrived.
6. The `send()` API echoes the data back to the client.
7. The `close()` API closes any open socket descriptors.
8. The `unlink()` API removes the UNIX path name from the file system.

Socket flow of events: Client application that uses AF_UNIX_CCSID address family

The second example uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the UNIX address family with the stream transport (`SOCK_STREAM`) is used for this socket. You can also use the `socketpair()` API to initialize a UNIX socket.
AF_UNIX or AF_UNIX_CCSID are the only address families to support the `socketpair()` API. The `socketpair()` API returns two socket descriptors that are unnamed and connected.
2. After the socket descriptor is received, the `connect()` API is used to establish a connection to the server.
3. The `send()` API sends 250 bytes of data that are specified in the server application with the `SO_RCVLOWAT` socket option.
4. The `recv()` API loops until all 250 bytes of the data have arrived.
5. The `close()` API closes any open socket descriptors.

Related reference:

“AF_UNIX_CCSID address family” on page 11

The AF_UNIX_CCSID family is compatible with the AF_UNIX address family and has the same limitations.

“Using AF_UNIX address family” on page 29

Sockets that use the AF_UNIX or AF_UNIX_CCSID address family can be connection-oriented (type `SOCK_STREAM`) or connectionless (type `SOCK_DGRAM`).

Related information:

Path name format

`recvfrom()`--Receive Data API

`accept()`--Wait for Connection Request and Make Connection API

`getpeername()`--Retrieve Destination Address of Socket API

`getsockname()`--Retrieve Local Address of Socket API

`recvmsg()`--Receive a Message Over a Socket API

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`bind()`--Set Local Address for Socket API

`unlink()`--Remove Link to File API

`listen()`--Invite Incoming Connections Requests API

`send()`--Send Data API

`connect()`--Establish Connection or Destination Address API

`recv()`--Receive Data API

`socketpair()`--Create a Pair of Sockets API

Example: Server application that uses AF_UNIX_CCSID address family:

This example program shows a server application that uses the AF_UNIX_CCSID address family.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* *****  
/* This example program provides code for a server application for      */  
/* AF_UNIX_CCSID address family.                                         */  
/* *****  
  
/* *****  
/* Header files needed for this sample program                          */  
/* *****
```

```

/*****/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*****/
/* Constants used by this program */
/*****/
#define SERVER_PATH    "/tmp/server"
#define BUFFER_LENGTH  250
#define FALSE          0

void main()
{
    /*****/
    /* Variable and structure definitions. */
    /*****/
    int    sd=-1, sd2=-1;
    int    rc, length;
    char   buffer[BUFFER_LENGTH];
    struct sockaddr_unc serveraddr;

    /*****/
    /* A do/while(FALSE) loop is used to make error cleanup easier. The */
    /* close() of each of the socket descriptors is only done once at the */
    /* very end of the program. */
    /*****/
    do
    {
        /*****/
        /* The socket() function returns a socket descriptor, which represents */
        /* an endpoint. The statement also identifies that the UNIX CCSID */
        /* address family with the stream transport (SOCK_STREAM) will be */
        /* used for this socket. */
        /*****/
        sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
        if (sd < 0)
        {
            perror("socket() failed");
            break;
        }

        /*****/
        /* After the socket descriptor is created, a bind() function gets a */
        /* unique name for the socket. */
        /*****/
        memset(&serveraddr, 0, sizeof(serveraddr));
        serveraddr.sunc_family      = AF_UNIX_CCSID;
        serveraddr.sunc_format      = SO_UNC_USE_QLG;
        serveraddr.sunc_qlg.CCSID   = 500;
        serveraddr.sunc_qlg.Path_Type = QLG_PTR_SINGLE;
        serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
        serveraddr.sunc_path.p_unix = SERVER_PATH;

        rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
        if (rc < 0)
        {
            perror("bind() failed");
            break;
        }

        /*****/
        /* The listen() function allows the server to accept incoming */
        /* client connections. In this example, the backlog is set to 10. */
        /* This means that the system will queue 10 incoming connection */
        /*****/

```

```

/* requests before the system starts rejecting the incoming */
/* requests. */
/*****/
rc = listen(sd, 10);
if (rc < 0)
{
    perror("listen() failed");
    break;
}

printf("Ready for client connect().\n");

/*****/
/* The server uses the accept() function to accept an incoming */
/* connection request. The accept() call will block indefinitely */
/* waiting for the incoming connection to arrive. */
/*****/
sd2 = accept(sd, NULL, NULL);
if (sd2 < 0)
{
    perror("accept() failed");
    break;
}

/*****/
/* In this example we know that the client will send 250 bytes of */
/* data over. Knowing this, we can use the SO_RCVLOWAT socket */
/* option and specify that we don't want our recv() to wake up */
/* until all 250 bytes of data have arrived. */
/*****/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                (char *)&length, sizeof(length));

if (rc < 0)
{
    perror("setsockopt(SO_RCVLOWAT) failed");
    break;
}

/*****/
/* Receive that 250 bytes data from the client */
/*****/
rc = recv(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("recv() failed");
    break;
}

printf("%d bytes of data were received\n", rc);
if (rc == 0 ||
    rc < sizeof(buffer))
{
    printf("The client closed the connection before all of the\n");
    printf("data was sent\n");
    break;
}

/*****/
/* Echo the data back to the client */
/*****/
rc = send(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("send() failed");
    break;
}

```

```

    }

    /*****
    /* Program complete
    *****/

} while (FALSE);

/*****
/* Close down any open socket descriptors
*****/
if (sd != -1)
    close(sd);

if (sd2 != -1)
    close(sd2);

/*****
/* Remove the UNIX path name from the file system
*****/
unlink(SERVER_PATH);
}

```

Example: Client application that uses AF_UNIX_CCSID address family:

This example program shows a client application that uses the AF_UNIX_CCSID address family.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This example program provides code for a client application for
/* AF_UNIX_CCSID address family.
*****/

/*****
/* Header files needed for this sample program
*****/
#include <stdio.h>
#include <string.h>
#include <wchar.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*****
/* Constants used by this program
*****/
#define SERVER_PATH    "/tmp/server"
#define BUFFER_LENGTH  250
#define FALSE          0

/* Pass in 1 parameter which is either the */
/* path name of the server as a UNICODE  */
/* string, or set the server path in the  */
/* #define SERVER_PATH which is a CCSID   */
/* 500 string.
*****/
void main(int argc, char *argv[])
{
    /*****
    /* Variable and structure definitions.
    *****/
    int    sd=-1, rc, bytesReceived;
    char    buffer[BUFFER_LENGTH];
    struct sockaddr_unc serveraddr;

```

```

/*****
/* A do/while(FALSE) loop is used to make error cleanup easier. The */
/* close() of the socket descriptor is only done once at the very end */
/* of the program. */
*****/
do
{
    /*****/
    /* The socket() function returns a socket descriptor, which represents */
    /* an endpoint. The statement also identifies that the UNIX CCSID */
    /* address family with the stream transport (SOCK_STREAM) will be */
    /* used for this socket. */
    /*****/
    sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
    if (sd < 0)
    {
        perror("socket() failed");
        break;
    }

    /*****/
    /* If an argument was passed in, use this as the server, otherwise */
    /* use the #define that is located at the top of this program. */
    /*****/
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sunc_family = AF_UNIX_CCSID;
    if (argc > 1)
    {
        /* The argument is a UNICODE path name. Use the default format */
        serveraddr.sunc_format = SO_UNC_DEFAULT;
        wcsncpy(serveraddr.sunc_path.wide, (wchar_t *) argv[1]);
    }
    else
    {
        /* The local #define is CCSID 500. Set the Qlg_Path_Name to use */
        /* the character format */
        serveraddr.sunc_format = SO_UNC_USE_QLG;
        serveraddr.sunc_qlg.CCSID = 500;
        serveraddr.sunc_qlg.Path_Type = QLG_CHAR_SINGLE;
        serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
        strcpy((char *)&serveraddr.sunc_path, SERVER_PATH);
    }

    /*****/
    /* Use the connect() function to establish a connection to the */
    /* server. */
    /*****/
    rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
    if (rc < 0)
    {
        perror("connect() failed");
        break;
    }

    /*****/
    /* Send 250 bytes of a's to the server */
    /*****/
    memset(buffer, 'a', sizeof(buffer));
    rc = send(sd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
        perror("send() failed");
        break;
    }

    /*****/
    /* In this example we know that the server is going to respond with */
    /* the same 250 bytes that we just sent. Since we know that 250 */

```



```

/* bytes are going to be sent back to us, we can use the */
/* SO_RCVLOWAT socket option and then issue a single recv() and */
/* retrieve all of the data. */
/*
/* The use of SO_RCVLOWAT is already illustrated in the server */
/* side of this example, so we will do something different here. */
/* The 250 bytes of the data may arrive in separate packets, */
/* therefore we will issue recv() over and over again until all */
/* 250 bytes have arrived. */
/*****/
bytesReceived = 0;
while (bytesReceived < BUFFER_LENGTH)
{
    rc = recv(sd, & buffer[bytesReceived],
              BUFFER_LENGTH - bytesReceived, 0);
    if (rc < 0)
    {
        perror("recv() failed");
        break;
    }
    else if (rc == 0)
    {
        printf("The server closed the connection\n");
        break;
    }

    /*****/
    /* Increment the number of bytes that have been received so far */
    /*****/
    bytesReceived += rc;
}

} while (FALSE);

/*****/
/* Close down any open socket descriptors */
/*****/
if (sd != -1)
    close(sd);
}

```

Advanced socket concepts

Advanced socket concepts go beyond a general discussion of what sockets are and how they work. These concepts provide ways to design socket applications for **larger and more complex networks**.

Asynchronous I/O

Asynchronous I/O APIs provide a method for **threaded client/server models** to perform **highly concurrent and memory-efficient I/O**.

In previous threaded client/server models, typically **two I/O models** have **prevailed**. The first model dedicates **one thread per client connection**. The first model **consumes too many threads** and **might incur a substantial sleep and wake-up cost**. The second model **minimizes the number of threads by issuing the select() API on a large set of client connections and delegating a readied client connection or request to a thread**. In the second model, you must **select or mark on each subsequent select, which might cause a substantial amount of redundant work**.

Asynchronous I/O and **overlapped I/O** resolve both these dilemmas by **passing data to and from user buffers after control has been returned to the user application**. Asynchronous I/O notifies these worker threads when **data is available to be read** or when a **connection has become ready to transmit data**.

Asynchronous I/O advantages

- Asynchronous I/O **uses system resources more efficiently**. Data copies from and to user buffers are **asynchronous to the application that initiates the request**. This overlapped processing **makes efficient use of multiple processors** and in many cases **improves paging rates because system buffers are freed for reuse when data arrives**.
- Asynchronous I/O **minimizes** process/thread **wait time**.
- Asynchronous I/O provides **immediate service to client requests**.
- Asynchronous I/O **decreases the sleep and wake-up cost** on average.
- Asynchronous I/O **handles bursty application** efficiently.
- Asynchronous I/O provides **better scalability**.
- Asynchronous I/O provides the most efficient method of handling large data transfers. The fillBuffer flag on the QsoStartRecv() API informs the operating system to acquire a large amount of data before completing the Asynchronous I/O. Large amounts of data can also be sent with one asynchronous operation.
- Asynchronous I/O minimizes the number of threads that are needed.
- Asynchronous I/O optionally can use timers to specify the maximum time allowed for this operation to complete asynchronously. Servers close a client connection if it has been idle for a set amount of time. The asynchronous timers allow the server to enforce this time limit.
- Asynchronous I/O initiates secure session asynchronously with the gsk_secure_soc_startInit() API.

Table 11. Asynchronous I/O APIs

API	Description
gsk_secure_soc_startInit()	Starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session. Note: This API supports only sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
gsk_secure_soc_startRecv()	Starts an asynchronous receive operation on a secure session. Note: This API supports only sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
gsk_secure_soc_startSend()	Starts an asynchronous send operation on a secure session. Note: This API supports only sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
QsoCreateIOCompletionPort()	Creates a common wait point for completed asynchronous overlapped I/O operations. The QsoCreateIOCompletionPort() API returns a port handle that represents the wait point. This handle is specified on the QsoStartRecv(), QsoStartSend(), QsoStartAccept(), gsk_secure_soc_startRecv(), or gsk_secure_soc_startSend() APIs to initiate asynchronous overlapped I/O operations. Also this handle can be used with QsoPostIOCompletion() to post an event on the associated I/O completion port.
QsoDestroyIOCompletionPort()	Destroys an I/O completion port.
QsoWaitForIOCompletionPort()	Waits for completed overlapped I/O operation. The I/O completion port represents this wait point.
QsoStartAccept()	Starts an asynchronous accept operation. Note: This API supports only sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.

Table 11. Asynchronous I/O APIs (continued)

API	Description
QsoStartRecv()	Starts an asynchronous receive operation. Note: This API supports only sockets with address family AF_INET or AF_INET6 and type SOCK_STREAM.
QsoStartSend()	Starts an asynchronous send operation. Note: This API supports only sockets with the AF_INET or AF_INET6 address families with the SOCK_STREAM socket type.
QsoPostIOCompletion()	Allows an application to notify a completion port that some API or activity has occurred.
QsoGenerateOperationId()	Allows an application to associate an operation identifier that is unique for a socket.
QsoIsOperationPending()	Allows an application to check if one or more asynchronous I/O operations are pending on the socket.
QsoCancelOperation()	Allows an application to cancel one or more asynchronous I/O operations that are pending on the socket.

How asynchronous I/O works

An application creates an I/O completion port using the QsoCreateIOCompletionPort() API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests. The application starts an input or an output function, specifying an I/O completion port handle. When the I/O is completed, status information and an application-defined handle is posted to the specified I/O completion port. The post to the I/O completion port wakes up exactly one of possibly many threads that are waiting. The application receives the following items:

- A buffer that was supplied on the original request
- The length of data that was processed to or from that buffer
- An indication of what type of I/O operation has been completed
- Application-defined handle that was passed on the initial I/O request

This application handle can be the socket descriptor identifying the client connection, or a pointer to storage that contains extensive information about the state of the client connection. Since the operation was completed and the application handle was passed, the worker thread determines the next step to complete the client connection. Worker threads that process these completed asynchronous operations can handle many different client requests and are not tied to just one. Because copying to and from user buffers occurs asynchronously to the server processes, the wait time for client request diminishes. This can be beneficial on systems where there are multiple processors.

Asynchronous I/O structure

An application that uses asynchronous I/O has the structure demonstrated by the following code fragment.

```
#include <qsoasync.h>
struct Qso_OverlappedIO_t
{
    Qso_DescriptorHandle_t descriptorHandle;
    void *buffer;
    size_t bufferLength;
    int postFlag : 1;
    int fillBuffer : 1;
    int postFlagResult : 1;
    int reserved1 : 29;
```

```

int returnValue;
int errnoValue;
int operationCompleted;
int secureDataTransferSize;
unsigned int bytesAvailable;
struct timeval operationWaitTime;
int postedDescriptor;
char reserved2[40];
}

```

Related reference:

“Example: Using asynchronous I/O” on page 116

An application creates an I/O completion port using the `QsoCreateIOCompletionPort()` API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests.

“Socket application design recommendations” on page 88

Before working with a socket application, assess the functional requirements, goals, and needs of the socket application. Also, consider the performance requirements and the system resource impacts of the application.

“Examples: Connection-oriented designs” on page 91

You can design a connection-oriented socket server on the system in a number of ways. These example programs can be used to create your own connection-oriented designs.

“Example: Using signals with blocking socket APIs” on page 164

When a process or an application becomes blocked, signals allow you to be notified. They also provide a time limit for blocking processes.

Related information:

`gsk_secure_soc_startInit()`--Start asynchronous operation to negotiate a secure session API

`gsk_secure_soc_startRecv()`--Start asynchronous receive operation on a secure session API

`gsk_secure_soc_startSend()`--Start asynchronous send operation on a secure session API

`QsoCreateIOCompletionPort()`--Create I/O Completion Port API

`QsoDestroyIOCompletionPort()`--Destroy I/O Completion Port API

`QsoWaitForIOCompletion()`--Wait for I/O Operation API

`QsoStartAccept()`--Start asynchronous accept operation API

`QsoStartSend()`--Start Asynchronous Send Operation API

`QsoStartRecv()`--Start Asynchronous Receive Operation API

`QsoPostIOCompletion()`--Post I/O Completion Request API

`QsoGenerateOperationId()`--Get an I/O Operation ID

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

`QsoCancelOperation()`--Cancel an I/O Operation

Secure sockets


Currently, you have two methods to create secure socket applications on the IBM i operating system. The SSL_ APIs and Global Security Kit (GSKit) APIs provide communications privacy over an open communications network, which in most cases is the Internet.

These APIs allow client/server applications to communicate in a way that prevents eavesdropping, tampering, and message forgery. Both the SSL_ APIs and Global Secure Toolkit (GSKit) APIs support server and client authentication, and both allow an application to use the Secure Sockets Layer (SSL) protocol. However, GSKit APIs are supported for all IBM systems, while the SSL_ APIs exist only in the IBM i operating system. To enhance portability across systems, it is suggested that you use GSKit APIs when developing applications for secure socket connections.

Overview of secure sockets

Originally developed by Netscape, the Secure Sockets Layer (SSL) protocol is a layered protocol to be used on top of a reliable transport, such as Transmission Control Protocol (TCP), to provide secure communications for an application. A few of the many applications that require secure communications are Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and Telnet.

An SSL-enabled application typically needs to use a port different from an application that is not SSL-enabled. For example, an SSL-enabled browser accesses an SSL-enabled HTTP Server with a Universal Resource Locator (URL) that begins https rather than http. In most cases, a URL of https attempts to open a connection to port 443 of the server system instead of to port 80 that the standard HTTP Server uses.

- | There are multiple versions of the SSL protocol defined. The latest version, Transport Layer Security (TLS) Version 1.2, provides an evolutionary upgrade from TLS Version 1.1. Both SSL_ APIs and the GSKit APIs support TLS Version 1.2, TLS Version 1.1, TLS Version 1.0, SSL Version 3.0, and SSL Version 2.0. For more information about TLS Version 1.2, see RFC 5246: "The Transport Layer Security (TLS) Protocol Version 1.2" .

Global Security Kit (GSKit) APIs

Global Security Kit (GSKit) is a set of programmable interfaces that allow an application to be SSL enabled.

Just like the SSL_ APIs, the GSKit APIs allow you to implement the SSL and Transport Layer Security (TLS) protocols in your socket application program. However, GSKit APIs are supported across IBM systems and are easier to program than SSL_ APIs. In addition, new GSKit APIs have been added to provide asynchronous capabilities for negotiating a secure session, sending secure data, and receiving secure data. These asynchronous APIs exist only in the IBM i operating system and cannot be ported to other systems.

GSKit on IBM i uses System SSL. For more information about the properties and attributes of System SSL, see System SSL Properties.

Note: The GSKit APIs only support sockets with an address family of AF_INET or AF_INET6 and type SOCK_STREAM.

The following table describes the GSKit APIs.

Table 12. Global Security Kit APIs

API	Description
<code>gsk_attribute_get_buffer()</code>	Obtains specific character string information about a secure session or an SSL environment, such as certificate store file, certificate store password, application ID, and ciphers.
<code>gsk_attribute_get_cert_info()</code>	Obtains specific information about either the server or client certificate for a secure session or an SSL environment.
<code>gsk_attribute_get_enum_value()</code>	Obtains values for specific enumerated data for a secure session or an SSL environment.
<code>gsk_attribute_get_numeric_value()</code>	Obtains specific numeric information about a secure session or an SSL environment.
<code>gsk_attribute_set_callback()</code>	Sets callback pointers to routines in the user application. The application can then use these routines for special purposes.

Table 12. Global Security Kit APIs (continued)

API	Description
<code>gsk_attribute_set_buffer()</code>	Sets a specified buffer attribute to a value inside the specified secure session or an SSL environment.
<code>gsk_attribute_set_enum()</code>	Sets a specified enumerated type attribute to an enumerated value in the secure session or SSL environment.
<code>gsk_attribute_set_numeric_value()</code>	Sets specific numeric information for a secure session or an SSL environment.
<code>gsk_environment_close()</code>	Closes the SSL environment and releases all storage associated with the environment.
<code>gsk_environment_init()</code>	Initializes the SSL environment after any required attributes are set.
<code>gsk_environment_open()</code>	Returns an SSL environment handle that must be saved and used on subsequent gsk calls.
<code>gsk_secure_soc_close()</code>	Closes a secure session and free all the associated resources for that secure session.
<code>gsk_secure_soc_init()</code>	Negotiates a secure session, using the attributes set for the SSL environment and the secure session.
<code>gsk_secure_soc_misc()</code>	Performs miscellaneous APIs for a secure session.
<code>gsk_secure_soc_open()</code>	Obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related API calls.
<code>gsk_secure_soc_read()</code>	Receives data from a secure session.
<code>gsk_secure_soc_startInit()</code>	Starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session.
<code>gsk_secure_soc_write()</code>	Writes data on a secure session.
<code>gsk_secure_soc_startRecv()</code>	Initiates an asynchronous receive operation on a secure session.
<code>gsk_secure_soc_startSend()</code>	Initiates an asynchronous send operation on a secure session.
<code>gsk_strerror()</code>	Retrieves an error message and associated text string that describes a return value that was returned from calling a GSKit API.

An application that uses the sockets and GSKit APIs contains the following elements:

1. A call to `socket()` to obtain a socket descriptor.
2. A call to `gsk_environment_open()` to obtain a handle to an SSL environment.
3. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the SSL environment. At a minimum, either a call to `gsk_attribute_set_buffer()` to set the `GSK_OS400_APPLICATION_ID` value or to set the `GSK_KEYRING_FILE` value. Only one of these should be set. It is preferred that you use the `GSK_OS400_APPLICATION_ID` value. Also ensure that you set the type of application (client or server), `GSK_SESSION_TYPE`, using `gsk_attribute_set_enum()`.
4. A call to `gsk_environment_init()` to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that run using this environment.

5. Socket calls to activate a connection. It calls `connect()` to activate a connection for a client program, or it calls `bind()`, `listen()`, and `accept()` to enable a server to accept incoming connection requests.
6. A call to `gsk_secure_soc_open()` to obtain a handle to a secure session.
7. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the secure session. At a minimum, a call to `gsk_attribute_set_numeric_value()` to associate a specific socket with this secure session.
8. A call to `gsk_secure_soc_init()` to initiate the SSL handshake negotiation of the cryptographic parameters.

Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server, which the client is connecting to, has enabled client authentication. The `gsk_attribute_set_buffer(GSK_OS400_APPLICATION_ID)` or `gsk_attribute_set_buffer(GSK_KEYRING_FILE)` API calls identify (though in dissimilar ways) the key database file, from which the certificate and private key that are used during the handshake are obtained.

9. Calls to `gsk_secure_soc_read()` and `gsk_secure_soc_write()` to receive and send data.
10. A call to `gsk_secure_soc_close()` to end the secure session.
11. A call to `gsk_environment_close()` to close the SSL environment.
12. A call to `close()` to destroy the connected socket.

Related reference:

“Example: GSKit secure server with asynchronous data receive” on page 123

This example demonstrates how to establish a secure server using Global Security Kit (GSKit) APIs.

“Example: GSKit secure server with asynchronous handshake” on page 133

The `gsk_secure_soc_startInit()` API allows you to create secure server applications that can handle requests asynchronously.

“Example: Establishing a secure client with Global Security Kit APIs” on page 143

This example demonstrates how to establish a client using the Global Security Kit (GSKit) APIs.

Related information:

`gsk_attribute_get_buffer()`--Get character information about a secure session or an SSL environment API

`gsk_attribute_get_cert_info()`--Get information about a local or partner certificate API

`gsk_attribute_get_enum()`--Get enumerated information about a secure session or an SSL environment API

`gsk_attribute_get_numeric_value()`--Get numeric information about a secure session or an SSL environment API

`gsk_attribute_set_callback()`--Set callback pointers to routines in the user application API

`gsk_attribute_set_buffer()`--Set character information for a secure session or an SSL environment API

`gsk_attribute_set_enum()`--Set enumerated information for a secure session or an SSL environment API

`gsk_attribute_set_numeric_value()`--Set numeric information for a secure session or an SSL environment API

`gsk_environment_close()`--Close an SSL environment API

`gsk_environment_init()`--Initialize an SSL environment API

`gsk_environment_open()`--Get a handle for an SSL environment API

`gsk_secure_soc_close()`--Close a secure session API

`gsk_secure_soc_init()`--Negotiate a secure session API

`gsk_secure_soc_misc()`--Perform miscellaneous functions for a secure session API

`gsk_secure_soc_open()`--Get a handle for a secure session API

`gsk_secure_soc_startInit()`--Start asynchronous operation to negotiate a secure session API

`gsk_secure_soc_read()`--Receive data on a secure session API

gsk_secure_soc_write()--Send data on a secure session API
 gsk_secure_soc_startRecv()--Start asynchronous receive operation on a secure session API
 gsk_secure_soc_startSend()--Start asynchronous send operation on a secure session API
 gsk_strerror()--Retrieve GSKit runtime error message API
 socket()--Create Socket API
 bind()--Set Local Address for Socket API
 connect()--Establish Connection or Destination Address API
 listen()--Invite Incoming Connections Requests API
 accept()--Wait for Connection Request and Make Connection API
 close()--Close File or Socket Descriptor API

SSL_ APIs

- | Do not use the SSL_ APIs for any new applications. They are not able to support or be aware of many of
- | the new features of System SSL. The documentation remains for existing applications that are not yet
- | converted to GSKit.

Unlike GSKit APIs, SSL_ APIs only exist in the IBM i operating system. The following table describes the SSL_ APIs that are supported in the IBM i implementation.

Table 13. SSL_ APIs

API	Description
SSL_Create()	Enable SSL support for the specified socket descriptor.
SSL_Destroy()	End SSL support for the specified SSL session and socket.
SSL_Handshake()	Initiate the SSL handshake protocol.
SSL_Init()	Initialize the current job for SSL and establish the SSL security information for the current job. Note: Either an SSL_Init() or SSL_Init_Application() API must be processed before SSL can be used.
SSL_Init_Application()	Initialize the current job for SSL and establish the SSL security information for the current job. Note: Either an SSL_Init() or SSL_Init_Application() API must be processed before SSL can be used.
SSL_Read()	Receive data from an SSL-enabled socket descriptor.
SSL_Write()	Write data to an SSL-enabled socket descriptor.
SSL_Strerror()	Retrieve SSL runtime error message.
SSL_Perror()	Print SSL error message.
QlgSSL_Init()	Initialize the current job for SSL and establish the SSL security information for the current job using NLS-enabled path name.

An application that uses the sockets and SSL_ APIs contains the following elements:

- A call to socket() to obtain a socket descriptor.
- Either call SSL_Init() or SSL_Init_Application() to initialize the job environment for SSL processing and to establish the SSL security information for all SSL sessions that run in the current job. Only one of these APIs should be used. It is preferred that you use the SSL_Init_Application() API.
- Socket calls to activate a connection. It calls connect() to activate a connection for a client program, or it calls bind(), listen(), and accept() to enable a server to accept incoming connection requests.
- A call to SSL_Create() to enable SSL support for the connected socket.

- A call to `SSL_Handshake()` to initiate the SSL handshake negotiation of the cryptographic parameters.

Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server, which the client is connecting to, has enabled client authentication. The `SSL_Init()` or `SSL_Init_Application()` APIs identify (though in dissimilar ways) the key database file, from which the certificate and private key that are used during the handshake are obtained.

- Calls to `SSL_Read()` and `SSL_Write()` to receive and send data.
- A call to `SSL_Destroy()` to disable SSL support for the socket.
- A call to `close()` to destroy the connected sockets.

Related information:

`socket()`--Create Socket API

`listen()`--Invite Incoming Connections Requests API

`bind()`--Set Local Address for Socket API

`connect()`--Establish Connection or Destination Address API

`accept()`--Wait for Connection Request and Make Connection API

`close()`--Close File or Socket Descriptor API

`SSL_Create()`--Enable SSL Support for the Specified Socket Descriptor API

`SSL_Destroy()`--End SSL Support for the Specified SSL Session API

`SSL_Handshake()`--Initiate the SSL Handshake Protocol API

`SSL_Init()`--Initialize the Current Job for SSL API

`SSL_Init_Application()`--Initialize the Current Job for SSL Processing Based on the Application Identifier API

`SSL_Read()`--Receive Data from an SSL-Enabled Socket Descriptor API

`SSL_Write()`--Write Data to an SSL-Enabled Socket Descriptor API

`SSL_Strerror()`--Retrieve SSL Runtime Error Message API

`SSL_Perror()`--Print SSL Error Message API

Secure socket API error code messages

To get the error code messages for the secure socket API, follow these steps.

1. From a command line, enter `DSPMSGD RANGE(XXXXXXX)`, where XXXXXXX is the message ID for the return code. For example, if the return code is 3, you can enter `DSPMSGD RANGE(CPDBCBC9)`.
2. Select 1 to display message text.

Table 14. Secure socket API error code messages

Return code	Message ID	Constant name
0	CPCBC80	GSK_OK
1	CPDCA1	GSK_INVALID_HANDLE
2	CPDBCBC3	GSK_API_NOT_AVAILABLE
3	CPDBCBC9	GSK_INTERNAL_ERROR
4	CPC3460	GSK_INSUFFICIENT_STORAGE
5	CPDBC95	GSK_INVALID_STATE
8	CPDBCBC2	GSK_ERROR_CERT_VALIDATION
107	CPDBC98	GSK_KEYFILE_CERT_EXPIRED
201	CPDCA4	GSK_NO_KEYFILE_PASSWORD

Table 14. Secure socket API error code messages (continued)

Return code	Message ID	Constant name
202	CPDBC85	GSK_KEYRING_OPEN_ERROR
301	CPD8CA5	GSK_CLOSE_FAILED
402	CPDBC81	GSK_ERROR_NO_CIPHERS
403	CPDBC82	GSK_ERROR_NO_CERTIFICATE
404	CPDBC84	GSK_ERROR_BAD_CERTIFICATE
405	CPDBC86	GSK_ERROR_UNSUPPORTED_CERTIFICATE_TYPE
406	CPDBC8A	GSK_ERROR_IO
407	CPD8CA3	GSK_ERROR_BAD_KEYFILE_LABEL
408	CPD8CA7	GSK_ERROR_BAD_KEYFILE_PASSWORD
409	CPDBC9A	GSK_ERROR_BAD_KEY_LEN_FOR_EXPORT
410	CPDBC8B	GSK_ERROR_BAD_MESSAGE
411	CPDBC8C	GSK_ERROR_BAD_MAC
412	CPDBC8D	GSK_ERROR_UNSUPPORTED
414	CPDBC84	GSK_ERROR_BAD_CERT
415	CPDBC8B	GSK_ERROR_BAD_PEER
417	CPDBC92	GSK_ERROR_SELF_SIGNED
420	CPDBC96	GSK_ERROR_SOCKET_CLOSED
421	CPDBC87	GSK_ERROR_BAD_V2_CIPHER
422	CPDBC87	GSK_ERROR_BAD_V3_CIPHER
428	CPDBC82	GSK_ERROR_NO_PRIVATE_KEY
501	CPD8CA8	GSK_INVALID_BUFFER_SIZE
502	CPE3406	GSK_WOULD_BLOCK
601	CPD8CAC	GSK_ERROR_NOT_SSLV3
602	CPD8CA9	GSK_MISC_INVALID_ID
701	CPD8CA9	GSK_ATTRIBUTE_INVALID_ID
702	CPD8CA6	GSK_ATTRIBUTE_INVALID_LENGTH
703	CPD8CAA	GSK_ATTRIBUTE_INVALID_ENUMERATION
705	CPD8CAB	GSK_ATTRIBUTE_INVALID_NUMERIC
6000	CPDBC97	GSK_OS400_ERROR_NOT_TRUSTED_ROOT
6001	CPDBC81	GSK_OS400_ERROR_PASSWORD_EXPIRED
6002	CPD8CC9	GSK_OS400_ERROR_NOT_REGISTERED
6003	CPD8CAD	GSK_OS400_ERROR_NO_ACCESS
6004	CPDBC88	GSK_OS400_ERROR_CLOSED
6005	CPD8CCB	GSK_OS400_ERROR_NO_CERTIFICATE_AUTHORITIES
6007	CPDBC84	GSK_OS400_ERROR_NO_INITIALIZE
6008	CPD8CAE	GSK_OS400_ERROR_ALREADY_SECURE
6009	CPD8CAF	GSK_OS400_ERROR_NOT_TCP
6010	CPDBC9C	GSK_OS400_ERROR_INVALID_POINTER
6011	CPDBC9B	GSK_OS400_ERROR_TIMED_OUT
6012	CPC8CBA	GSK_OS400_ASYNCHRONOUS_RECV

Table 14. Secure socket API error code messages (continued)

Return code	Message ID	Constant name
6013	CPCBCBB	GSK_OS400_ASYNCHRONOUS_SEND
6014	CPDBCBC	GSK_OS400_ERROR_INVALID_OVERLAPPEDIO_T
6015	CPDBCBD	GSK_OS400_ERROR_INVALID_IOCOMPLETIONPORT
6016	CPDBCBE	GSK_OS400_ERROR_BAD_SOCKET_DESCRIPTOR
6017	CPDBCBF	GSK_OS400_ERROR_CERTIFICATE_REVOKED
6018	CPDBC87	GSK_OS400_ERROR_CRL_INVALID
6019	CPCBC88	GSK_OS400_ASYNCHRONOUS_SOC_INIT
0	CPCBC80	Successful return
-1	CPDBC81	SSL_ERROR_NO_CIPHERS
-2	CPDBC82	SSL_ERROR_NO_CERTIFICATE
-4	CPDBC84	SSL_ERROR_BAD_CERTIFICATE
-6	CPDBC86	SSL_ERROR_UNSUPPORTED_CERTIFICATE_TYPE
-10	CPDBC8A	SSL_ERROR_IO
-11	CPDBC8B	SSL_ERROR_BAD_MESSAGE
-12	CPDBC8C	SSL_ERROR_BAD_MAC
-13	CPDBC8D	SSL_ERROR_UNSUPPORTED
-15	CPDBC84	SSL_ERROR_BAD_CERT (map to -4)
-16	CPDBC8B	SSL_ERROR_BAD_PEER (map to -11)
-18	CPDBC92	SSL_ERROR_SELF_SIGNED
-21	CPDBC95	SSL_ERROR_BAD_STATE
-22	CPDBC96	SSL_ERROR_SOCKET_CLOSED
-23	CPDBC97	SSL_ERROR_NOT_TRUSTED_ROOT
-24	CPDBC98	SSL_ERROR_CERT_EXPIRED
-26	CPDBC9A	SSL_ERROR_BAD_KEY_LEN_FOR_EXPORT
-91	CPDBCBI	SSL_ERROR_KEYPASSWORD_EXPIRED
-92	CPDBCBI	SSL_ERROR_CERTIFICATE_REJECTED
-93	CPDBCBI	SSL_ERROR_SSL_NOT_AVAILABLE
-94	CPDBCBI	SSL_ERROR_NO_INIT
-95	CPDBCBI	SSL_ERROR_NO_KEYRING
-97	CPDBCBI	SSL_ERROR_BAD_CIPHER_SUITE
-98	CPDBCBI	SSL_ERROR_CLOSED
-99	CPDBCBI	SSL_ERROR_UNKNOWN
-1009	CPDBCC9	SSL_ERROR_NOT_REGISTERED
-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPBCD8	SSL_ERROR_NO_REUSE

Related reference:

“Examples: Establishing secure connections” on page 123

You can create secure server and clients using either the Global Security Kit (GSKit) APIs or the Secure Sockets Layer (SSL_) APIs.

Client SOCKS support

The IBM i operating system supports SOCKS version 4. This enables programs that use the AF_INET address family with the SOCK_STREAM socket type to communicate with server programs running on systems outside a firewall.

A firewall is a very secure host that a network administrator places between a secure internal network and a less secure external network. Typically such a network configuration does not allow communications that originate from the secure host to be routed on the less secure network, and vice versa. Proxy servers that exist on the firewall help manage required access between secure hosts and less secure networks.

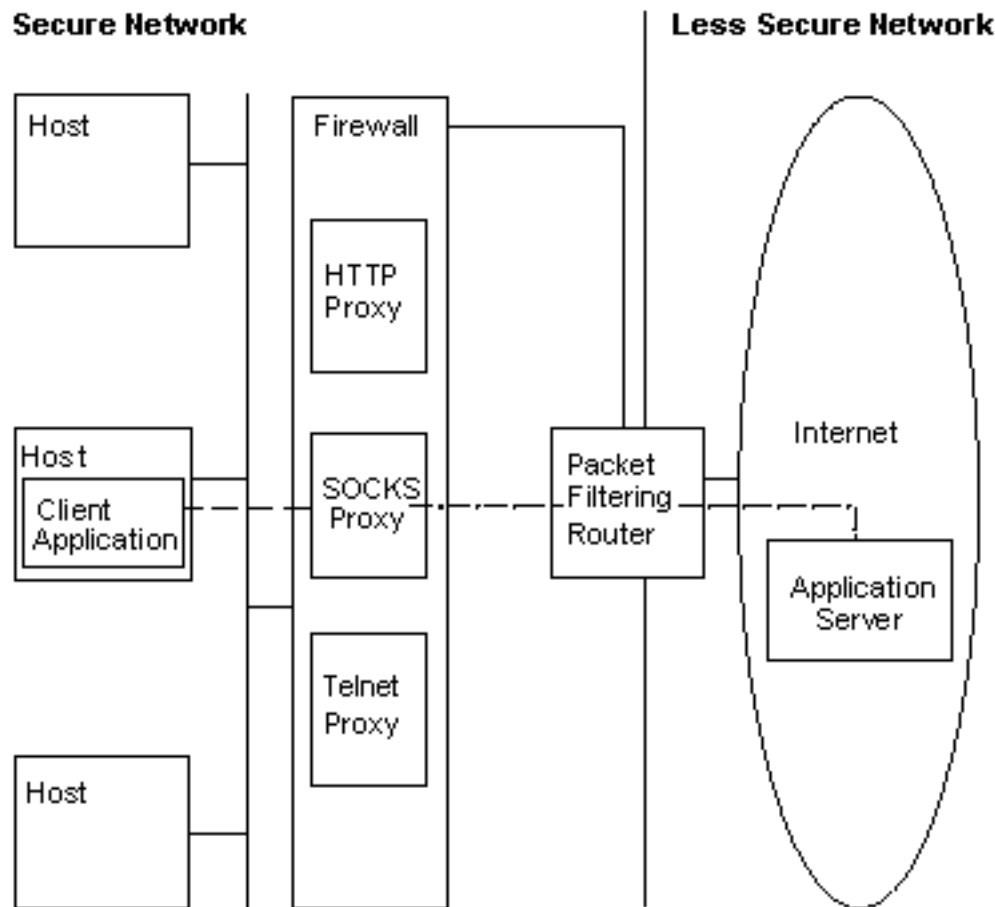
Applications that run on hosts in a secure internal network must send their requests to firewall proxy servers to navigate the firewall. The proxy servers can then forward these requests to the real server on the less secure network and relay the reply back to the applications on the originating host. A common example of a proxy server is an HTTP proxy server. Proxy servers perform a number of tasks for HTTP clients:

- They hide your internal network from outside systems.
- They protect the host from direct access by outside systems.
- They can filter data that comes in from outside if they are properly designed and configured.

HTTP proxy servers handle only HTTP clients.

A common alternative to running multiple proxy servers on a firewall is to run a more robust proxy server known as a SOCKS server. A SOCKS server can act as a proxy for any TCP client connection that is established through the sockets API. The key advantage of the IBM i Client SOCKS support is that it enables client applications to access a SOCKS server transparently without changing any client code.

The following figure shows a common firewall arrangement with an HTTP proxy, a telnet proxy, and a SOCKS proxy on the firewall. Notice that the two separate TCP connections used for the secure client that is accessing a server on the internet. One connection leads from the secure host to the SOCKS server, and the other leads from the less secure network to the SOCKS server.



Legend:

Secure TCP Connection - - - - -
 Less Secure TCP Connection - . - . -
 Local Area Network _____

Two actions are required on the secure client host to use a SOCKS server:

1. Configure a SOCKS server.
2. On the secure client system, define all outbound client TCP connections that are to be directed to the SOCKS server on the client system.

To configure client SOCKS support, follow these steps:

- a. From System i® Navigator, expand *your system* > **Network** > **TCP/IP Configuration**.
- b. Right-click **TCP/IP Configuration**.
- c. Click **Properties**.
- d. Click the **SOCKS** tab.
- e. Enter your connection information about the SOCKS page.

Note: The secure client SOCKS configuration data is saved in the QASOSCFG file in library QUSRSYS on the secure client host system.

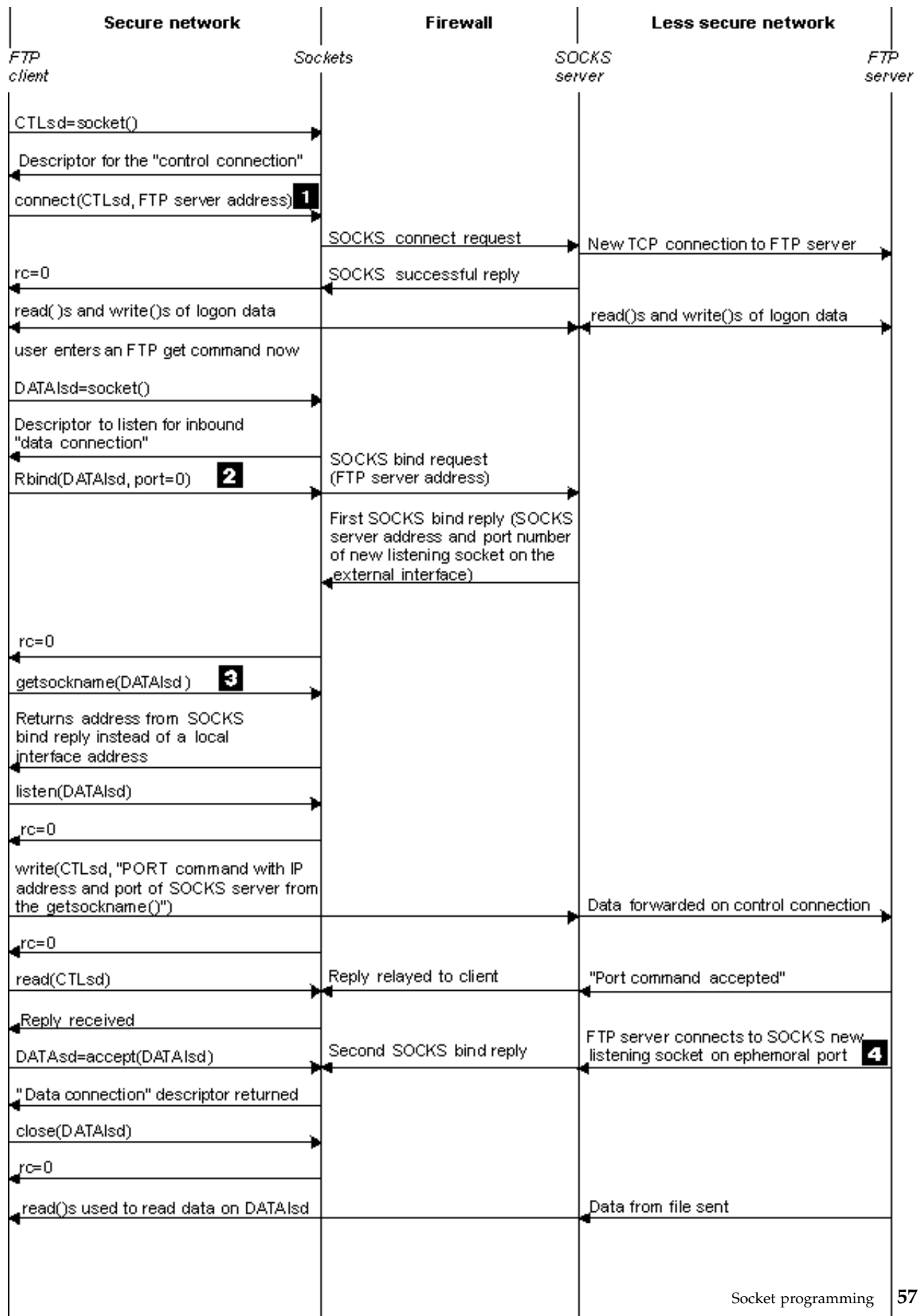
When configured, the system automatically directs certain outbound connections to the SOCKS server you specified on the SOCKS page. You do not need to make any changes to the secure client application.

When it receives the request, the SOCKS server establishes a separate external TCP/IP connection to the server in the less secure network. The SOCKS server then relays data between the internal and external TCP/IP connections.

Note: The remote host on the less secure network connects directly to the SOCKS server. It does not have direct access to the secure client.

Up to this point, *outbound* TCP connections that originate from the secure client have been addressed. Client SOCKS support also lets you tell the SOCKS server to allow an inbound connection request across a firewall. An Rbind() call from the secure client system allows this communication. For Rbind() to operate, the secure client must have previously issued a connect() call and the call must have resulted in an outbound connection over the SOCKS server. The Rbind() inbound connection must be from the same IP address that was addressed by the outbound connection that the connect() established.

The following figure shows a detailed overview of how sockets APIs interact with a SOCKS server transparent to the application. In the example, the FTP client calls the Rbind() API instead of a bind() API, because the FTP protocol allows the FTP server to establish a data connection when there is a request from the FTP client to send files or data. It makes this call by recompiling the FTP client code with the __Rbind preprocessor #define, which defines bind() to be Rbind(). Alternatively, an application can explicitly code Rbind() in the pertinent source code. If an application does not require inbound connections across a SOCKS server, Rbind() should not be used.



Notes:

1. The FTP client initiates an outbound TCP connection to a less secure network through a SOCKS server. The destination address that the FTP client specifies on the connect() is the IP address and port of the FTP server located on the less secure network. The secure host system is configured through the SOCKS page to direct this connection through the SOCKS server. When configured, the system automatically directs the connection to the SOCKS server that was specified through the SOCKS page.
2. A socket is opened and Rbind() is called to establish an inbound TCP connection. When established, this inbound connection is from the same destination-outbound IP address that was specified above. You must pair outbound and inbound connections over the SOCKS server for a particular thread. In other words, all Rbind() inbound connections should immediately follow the outbound connection over the SOCKS server. You cannot attempt to intervene non-SOCKS connections relating to this thread before the Rbind() runs.
3. getsockname() returns the SOCKS server address. The socket logically binds to the SOCKS server IP address coupled with a port that is selected through the SOCKS server. In this example, the address is sent through the "control connection" Socket CTled to the FTP server that is located on the less secure network. This is the address to which the FTP server connects. The FTP server connects to the SOCKS server and not directly to the secure host.
4. The SOCKS server establishes a data connection with the FTP client and relays data between the FTP client and the FTP server. Many SOCKS servers allow a fixed length of time for the server to connect to the Secure client. If the server does not connect within this time, errno ECONNABORTED is encountered on the accept().

Related information:

bind()--Set Local Address for Socket API

connect()--Establish Connection or Destination Address API

accept()--Wait for Connection Request and Make Connection API

getsockname()--Retrieve Local Address of Socket API

Rbind()--Set Remote Address for Socket API

Thread safety

A function is considered threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe only if all the functions it calls are also threadsafe. Socket APIs consist of system and network functions, which are both threadsafe.

All network functions with names that end in "_r" have similar semantics and are also threadsafe.

The other resolver routines are threadsafe with each other but they use the _res data structure. This data structure is shared between all threads in a process and can be changed by an application during a resolver call.

Related reference:

"Example: Using gethostbyaddr_r() for threadsafe network routines" on page 150

This example program uses the gethostbyaddr_r() API. All other routines with names that end in _r have similar semantics and are also threadsafe.

"Example: Updating and querying DNS" on page 172

This example shows how to query and update Domain Name System (DNS) records.

Nonblocking I/O

When an application issues one of the socket input APIs and there is no data to read, the **API blocks** and does **not return until there is data to read**.

Similarly, an application can block on a socket output API when data cannot be sent immediately. Finally, `connect()` and `accept()` can block while waiting for connection establishment with the partner's programs.

Sockets provide a method that enables application programs to issue APIs that block so that the API returns without delay. This is done by either calling `fcntl()` to turn on the `O_NONBLOCK` flag, or calling `ioctl()` to turn on the `FIONBIO` flag. When running in this nonblocking mode, if an API cannot be completed without blocking, it returns immediately. A `connect()` might return with `[EINPROGRESS]`, which means that the connection initiation has been started. You can then use the `poll()` or `select()` to determine when the connection has been completed. For all other APIs that are affected by running in the nonblocking mode, an error code of `[EWOULDBLOCK]` indicates that the call was unsuccessful.

You can use nonblocking with the following socket APIs:

- `accept()`
- `connect()`
- `gsk_secure_soc_read()`
- `gsk_secure_soc_write()`
- `read()`
- `readv()`
- `recv()`
- `recvfrom()`
- `recvmsg()`
- `send()`
- `send_file()`
- `send_file64()`
- `sendmsg()`
- `sendto()`
- `SSL_Read()`
- `SSL_Write()`
- `write()`
- `writenv()`

Related reference:

"Example: Nonblocking I/O and `select()`" on page 152

This sample program illustrates a server application that uses nonblocking and the `select()` API.

Related information:

`fcntl()`--Perform File Control Command API

`accept()`--Wait for Connection Request and Make Connection API

`ioctl()`--Perform I/O Control Request API

`recv()`--Receive Data API

`send()`--Send Data API

`connect()`--Establish Connection or Destination Address API

`gsk_secure_soc_read()`--Receive data on a secure session API

`gsk_secure_soc_write()`--Send data on a secure session API

`SSL_Read()`--Receive Data from an SSL-Enabled Socket Descriptor API

`SSL_Write()`--Write Data to an SSL-Enabled Socket Descriptor API

`read()`--Read from Descriptor API

`readv()`--Read from Descriptor Using Multiple Buffers API

`recvfrom()`--Receive Data API

recvmsg()--Receive a Message Over a Socket API
send_file()--Send a File over a Socket Connection API
send_file64()
--Send a Message Over a Socket API
sendto()--Send Data API
write()--Write to Descriptor API
writev()--Write to Descriptor Using Multiple Buffers API

Signals

An application program can **request to be notified asynchronously** (request that the system send a **signal**) when a condition that the application is interested in occurs.

There are **two asynchronous signals** that **sockets send to an application**.

1. **SIGURG** is a signal that is sent when **out-of-band (OOB) data is received on a socket** for which the concept of OOB data is supported. For example, a socket with an address family of AF_INET6 and a type of SOCK_STREAM can be conditioned to send a SIGURG signal.
2. **SIGIO** is a signal that is sent when **normal data, OOB data, error conditions**, or just about **anything happens on any type of socket**.

The application should ensure that it is able to handle receiving a signal before it requests the system to send signals. This is done by setting up **signal handlers**. One way to set a signal handler is by issuing the **sigaction() call**.

An application requests the system to send the **SIGURG** signal by one of the following methods:

- Issuing a **fcntl()** call and specifying a **process ID** or a **process group ID** with the **F_SETOWN** command.
- Issuing an **ioctl()** call and specifying the **FIOSETOWN** or the **SIOCSPGRP** command (request).

An application requests the system to send the **SIGIO** signal in two steps. First it must set the process ID or the process group ID as previously described for the **SIGURG** signal. This is to inform the system of where the application wants the signal to be delivered. Second, the application must do either of the following tasks:

- Issue the **fcntl()** call and specify the **F_SETFL** command with the **FASYNC** flag.
- Issue the **ioctl()** call and specify the **FIOASYNC** command.

This step requests the system to generate the **SIGIO** signal. Note that these steps can be done in any order. Also note that if an application issues these requests on a listening socket, the values set by the requests are inherited by all sockets that are returned to the application from the **accept()** API. That is, newly accepted sockets also have the same process ID or process group ID as well as the same information with regard to sending the **SIGIO** signal.

A socket can also generate synchronous signals on error conditions. Whenever an application receives [EPIPE] an **errno** on a socket API, a **SIGPIPE** signal is delivered to the process that issued the operation receiving the **errno** value. On a Berkeley Socket Distribution (BSD) implementation, by default the **SIGPIPE** signal ends the process that received the **errno** value. To remain compatible with previous releases of the IBM i implementation, the IBM i implementation uses a default behavior of ignoring for the **SIGPIPE** signal. This ensures that existing applications are not negatively affected by the addition of the signals API.

When a signal is delivered to a process that is blocked on a sockets API, the API returns from the wait with the **[EINTR] errno** value, allowing the application's signal handler to run. The APIs for which this occur are:

- **accept()**

- connect()
- poll()
- read()
- readv()
- recv()
- recvfrom()
- recvmsg()
- select()
- send()
- sendto()
- sendmsg()
- write()
- writev()

It is important to note that signals do not provide the application program with a socket descriptor that identifies where the condition being signalled actually exists. Thus, if the application program is using multiple socket descriptors, it must either poll the descriptors or use the select() call to determine why the signal was received.

Related concepts:

“Out-of-band data” on page 63

Out-of-band (OOB) data is **user-specific data** that **only has meaning for connection-oriented (stream) sockets**.

Related reference:

“Example: Using signals with blocking socket APIs” on page 164

When a process or an application becomes blocked, signals allow you to be notified. They also provide a time limit for blocking processes.

Related information:

accept()--Wait for Connection Request and Make Connection API

--Send a Message Over a Socket API

sendto()--Send Data API

write()--Write to Descriptor API

writev()--Write to Descriptor Using Multiple Buffers API

read()--Read from Descriptor API

readv()--Read from Descriptor Using Multiple Buffers API

connect()--Establish Connection or Destination Address API

recvfrom()--Receive Data API

recvmsg()--Receive a Message Over a Socket API

recv()--Receive Data API

send()--Send Data API

select()--Wait for Events on Multiple Sockets API

IP multicasting

IP multicasting allows an application to send a single IP datagram that a group of hosts in a network can receive.

The hosts that are in the group can reside on a single subnet or on different subnets that multicast-capable routers connect. Hosts can join and leave groups at any time. There are no restrictions on the location or number of members in a host group. For AF_INET, a class D IP address in the range

224.0.0.1 to 239.255.255.255 identifies a host group. For AF_INET6, an IPv6 address starting with FF00::/8 identifies the address as a multicast address. Refer to RFC 3513: "Internet Protocol Version 6 (IPv6)

Addressing Architecture"  for more information.

You can currently use IP multicasting with AF_INET and AF_INET6 address families.

An application program can send or receive multicast datagrams using the Sockets API and connectionless, SOCK_DGRAM type sockets. Multicasting is a one-to-many transmission method. Connection-oriented sockets of type SOCK_STREAM cannot be used for multicasting. When a socket of type SOCK_DGRAM is created, an application can use the setsockopt() API to control the multicast characteristics associated with that socket. The setsockopt() API accepts the following IPPROTO_IP level flags:

- IP_ADD_MEMBERSHIP: Joins the multicast group specified
- IP_DROP_MEMBERSHIP: Leaves the multicast group specified
- IP_MULTICAST_IF: Sets the interface over which outgoing multicast datagrams should be sent
- IP_MULTICAST_TTL: Sets the Time To Live (TTL) in the IP header for outgoing multicast datagrams
- IP_MULTICAST_LOOP: Specifies whether 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

The setsockopt() API also accepts the following IPPROTO_IPv6 level flags:

- IPv6_MULTICAST_IF: Sets the interface over which outgoing multicast datagrams are sent
- IPv6_MULTICAST_HOPS: Sets the hop limit values that are used for subsequent multicast packets sent by a socket
- IPv6_MULTICAST_LOOP: Specifies whether 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
- IPv6_JOIN_GROUP: Joins the multicast group specified
- IPv6_LEAVE_GROUP: Leaves the multicast group specified

Related reference:

"Examples: Using multicasting with AF_INET" on page 167

With IP multicasting, an application can send a single IP datagram that a group of hosts in a network can receive.

Related information:

setsockopt()--Set Socket Options API

File data transfer—send_file() and accept_and_recv()

IBM i sockets provide the send_file() and accept_and_recv() APIs that enable faster and easier file transfers over connected sockets.

These two APIs are especially useful for file-serving applications such as Hypertext Transfer Protocol (HTTP) servers.

The send_file() API enables the sending of file data directly from a file system over a connected socket with a single API call.

The accept_and_recv() API is a combination of three socket APIs: accept(), getsockname(), and recv().

Related reference:

"Examples: Transferring file data using send_file() and accept_and_recv() APIs" on page 176

These examples enable a server to communicate with a client by using the send_file() and accept_and_recv() APIs.

Related information:

send_file()--Send a File over a Socket Connection API
accept_and_recv()

Out-of-band data

Out-of-band (OOB) data is user-specific data that only has meaning for **connection-oriented (stream)** sockets.

Stream data is generally **received in the same order it is sent**. OOB data is **received independent of its position in the stream** (independent of the order in which it was sent). This is possible because the data is marked in such a way that, when it is sent from program A to program B, program B is **notified of its arrival**.

OOB data is supported on **AF_INET (SOCK_STREAM)** and **AF_INET6 (SOCK_STREAM)** only.

OOB data is sent by specifying the **MSG_OOB flag** on the send(), sendto(), and sendmsg() APIs.

The transmission of OOB data is the same as the transmission of regular data. It is sent after any data that is buffered. In other words, OOB data does not take precedence over any data that might be buffered; data is transmitted in the order that it was sent.

On the receiving side, things are a little more complex:

- The sockets API keeps track of OOB data that is received on a system by using an OOB marker. The OOB marker points to the last byte in the OOB data that was sent.

Note: The value that indicates which byte the OOB marker points to is set on a system basis (all applications use that value). This value must be consistent between the local and remote ends of a TCP connection. Socket applications that use this value must use it consistently between the client and server applications.

The SIOCATMARK ioctl() request determines if the read pointer is pointing to the last OOB byte.

Note: If multiple occurrences of OOB data are sent, the OOB marker points to the last OOB byte of the final OOB data occurrence.

- Independent of whether OOB data is received inline, an input operation processes data up to the OOB marker, if OOB data was sent.
- A recv(), recvmsg(), or recvfrom() API (with the MSG_OOB flag set) is used to receive OOB data. An error of [EINVAL] is returned if one of the receive APIs has been completed and one of the following situations occurs:
 - The socket option SO_OOBINLINE is not set and there is no OOB data to receive.
 - The socket option SO_OOBINLINE is set.

If the socket option SO_OOBINLINE is not set, and the sending program sent OOB data with a size greater than one byte, all the bytes but the last are considered normal data. (Normal data means that the receiving program can receive data without specifying the MSG_OOB flag.) The last byte of the OOB data that was sent is not stored in the normal data stream. This byte can only be retrieved by issuing a recv(), recvmsg(), or recvfrom() API with the MSG_OOB flag set. If a receive operation is issued with the MSG_OOB flag not set, and normal data is received, the OOB byte is deleted. Also, if multiple occurrences of OOB data are sent, the OOB data from the preceding occurrence is lost, and the position of the OOB data of the final OOB data occurrence is remembered.

If the socket option SO_OOBINLINE is set, then all of the OOB data that was sent is stored in the normal data stream. Data can be retrieved by issuing one of the three receive APIs without specifying the MSG_OOB flag (if it is specified, an error of [EINVAL] is returned). OOB data is not lost if multiple occurrences of OOB data are sent.

- OOB data is not discarded if SO_OOBINLINE is not set, OOB data has been received, and the user then sets SO_OOBINLINE on. The initial OOB byte is considered normal data.

- If SO_OOBLINE is not set, OOB data was sent, and the receiving program issued an input API to receive the OOB data, then the OOB marker is still valid. The receiving program can still check if the read pointer is at the OOB marker, even though the OOB byte was received.

Related concepts:

“Signals” on page 60

An application program can request to be notified asynchronously (request that the system send a *signal*) when a condition that the application is interested in occurs.

Related information:

--Send a Message Over a Socket API

Change TCP Attributes (CHGTCPA) command

I/O multiplexing—select()

Because asynchronous I/O provides a more efficient way to maximize your application resources, it is recommended that you use asynchronous I/O APIs rather than the select() API. However, your specific application design might allow select() to be used.

Like asynchronous I/O, the select() API creates a common point to wait for multiple conditions at the same time. However, select() allows an application to specify sets of descriptors to see if the following conditions exist:

- There is data to be read.
- Data can be written.
- An exception condition is present.

The descriptors that can be specified in each set can be socket descriptors, file descriptors, or any other object that is represented by a descriptor.

The select() API also allows the application to specify if it wants to wait for data to become available. The application can specify how long to wait.

Related reference:

“Example: Nonblocking I/O and select()” on page 152

This sample program illustrates a server application that uses nonblocking and the select() API.

Socket network functions

Socket network functions allow application programs to obtain information from the host, protocol, service, and network files.

The information can be accessed by name or by address, or by sequential access of the file. These network functions (or routines) are required when setting up communications between programs that run across networks, and thus are not used by AF_UNIX sockets.

The routines are as follows:

- Map host names to network addresses.
- Map network names to network numbers.
- Map protocol names to protocol numbers.
- Map service names to port numbers.
- Convert the byte order of Internet network addresses.
- Convert IP address and dotted decimal notation.

Included in the network routines is a group of routines called resolver routines. These routines make, send, and interpret packets for name servers in the Internet domain and are also used to do name

resolution. The resolver routines normally get called by `gethostbyname()`, `gethostbyaddr()`, `getnameinfo()`, and `getaddrinfo()` but can be called directly. Primarily resolver routines are used for accessing Domain Name System (DNS) through socket application.

Related concepts:

“Socket characteristics” on page 6

Sockets share some common characteristics.

Related reference:

“Example: Using `gethostbyaddr_r()` for threadsafe network routines” on page 150

This example program uses the `gethostbyaddr_r()` API. All other routines with names that end in `_r` have similar semantics and are also threadsafe.

“Domain Name System support”

The operating system provides applications with access to the Domain Name System (DNS) through the resolver functions.

Related information:

Sockets System Functions

`gethostbyname()`--Get Host Information for Host Name API

`getaddrinfo()`--Get Address Information API

`gethostbyaddr()`--Get Host Information for IP Address API

`getnameinfo()`--Get Name Information for Socket Address API

Domain Name System support

The operating system provides applications with access to the Domain Name System (DNS) through the resolver functions.

The DNS has the following three major components:

Domain name space and resource records

Specifications for a tree-structured name space and the data associated with the names.

Name servers


Server programs that hold information about the domain tree structure and set information.

Resolvers

Programs that extract information from name servers in response to client requests.

The resolvers provided in the IBM i implementation are socket functions that provide communication with a name server. These routines are used to make, send, update, and interpret packets, and perform name caching for performance. They also provide function for ASCII to EBCDIC and EBCDIC to ASCII conversion. Optionally, the resolver uses transaction signatures (TSIG) to securely communicate with the DNS.

For more information about domain names, see the following RFCs, which you can locate from the RFC

Search Engine  page.

- RFC 1034: Domain names - concepts and facilities.
- RFC 1035: Domain names - implementation and specification.
- RFC 1886: DNS Extensions to support IP version 6.
- RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE).
- RFC 2181: Clarifications to the DNS Specification.
- RFC 2845: Secret Key Transaction Authentication for DNS (TSIG).
- RFC 3152: DNS Delegation of IP6.ARPA.

Related reference:

“Socket network functions” on page 64

Socket network functions allow application programs to obtain information from the host, protocol, service, and network files.

Related information:

Domain Name System

Sockets System Functions

Environment variables

You can use environment variables to override default initialization of resolver functions.

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 are ignored.

Note: The name of the environment variable must be capitalized. The string value might be mixed case. Japanese systems using CCSID 290 should use uppercase characters and numbers only in both environment variables names and values. The list contains the descriptions of environment variables that can be used with the `res_init()` and `res_ninit()` APIs.

LOCALDOMAIN

Set this environment variable to a space-separated list of up to six search domains with a total of 256 characters (including spaces). This overrides the configured search list (`struct state.defdname` and `struct state.dnsrch`). If a search list is specified, the default local domain is not used on queries.

RES_OPTIONS

The `RES_OPTIONS` environment variable 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 that must appear in a name given to `res_query()` before an initial absolute query is made. The default for `n` is 1, meaning that if there are any dots in a name, the name is tried first as an absolute name before any search list elements are appended to it.
- **TIMEOUT: n** Sets the amount of time (in seconds) that the resolver waits for a response from a remote name server before giving up and trying the query again.
- **ATTEMPTS: n** Sets the number of queries that the resolver sends to a given nameServer before giving up and trying the next listed name server.
- **ROTATE:** Sets `RES_ROTATE` in `_res.options`, which rotates the 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

Set this environment variable to a space separated list of resolver option flags. This overrides the `RES_DEFAULT` options (`struct state.options`) and system configured values (Change TCP/IP Domain - `CHGTCPDMN`). The state options structure is initialized normally, using `RES_DEFAULT`, `OPTIONS` environment values and `CHGTCPDMN` configured values. Then this environment variable is used to override those defaults. The flags named in this environment variable might 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`, use the following command from a character-based interface:


```
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

Set this environment variable to a space-separated list of up to ten IP addresses/mask pairs in dotted decimal format (9.5.9.0/255.255.255.0) to create a sort list (struct state.sort_list).

Related information:

res_init()

res_ninit()

res_query()

Data caching

Data caching of responses to Domain Name System (DNS) queries is done by IBM i sockets in an effort to lessen the amount of network traffic. The cache is added to and updated as needed.

If RES_AAONLY (authoritative answers only) is set in _res.options, the query is always sent on the network. In this case, the cache is never checked for the answer. If RES_AAONLY is not set, the cache is checked for an answer to the query before any attempt to send it on the network is performed. If the answer is found and the time to live has not expired, the answer is returned to the user as the answer to the query. If the time to live has expired, the entry is removed, and the query is sent on the network. Also, if the answer is not found in the cache, the query is sent on the network.

Answers from the network are cached if the responses are authoritative. Nonauthoritative answers are not cached. Also, responses received as a result of an inverse query are not cached. You can clear this cache by updating the DNS configuration with either the Change TCP/IP Domain (CHGTCPDMN) command, Configure TCP/IP (CFGTCP) command, or through System i Navigator.

Related reference:

“Example: Updating and querying DNS” on page 172

This example shows how to query and update Domain Name System (DNS) records.

Domain Name System Security Extensions (DNSSEC)

| The original DNS protocol did not support security, making DNS vulnerable to attacks such as packet interception, spoofing, and cache poisoning, potentially compromising all future communications to a host. DNSSEC provides a means to secure DNS data by using digital signatures and public key cryptography.


| DNSSEC allows a resolver or name server to verify the authenticity and integrity of DNS response data by establishing a “chain of trust” to the source of the DNS data and validating the digital signatures.

| The main function of DNSSEC is to protect the user from forged data.

- | • Validate the origin of a DNS response
 - | – Trust that the data came from the expected source
- | • Validate the integrity of a DNS response
 - | – Trust that the data itself is correct
- | • Validate denial of existence
 - | – Trust a “no records to return” response

| DNSSEC does not provide any of the following functions:

- | • Encryption of data (for example, SSL)
- | • Protection from denial of service attacks

- Protection from going to phishing sites
- DNSSEC support in the IBM i resolver can be enabled by using the Change TCP/IP Domain (CHGTCPDMN) command. In DNSSEC terms, the IBM i resolver is a non-validating security-aware stub resolver. This means that when DNSSEC is enabled, the IBM i resolver sets the DNSSEC OK bit in its query messages to indicate that it can handle DNSSEC fields in responses. However, it relies on the name server to do the actual authentication and validation of the DNS response data. This dependency implies that to have a secure DNS solution, IBM i must trust the name server and also have a secure communication channel to the name server. One option to secure the communication channel is to configure the DNS server on the same partition as the resolver and have them communicate via the loopback address (127.0.0.1 for IPv4 or ::1 for IPv6). Another option is to use IP Security (IPSec) to secure the communication channel between IBM i and the name server.
- For more information about the IBM i DNS server, see Domain Name System.
- For more information about DNSSEC, see the following RFCs, which you can locate from the RFC Search Engine  page.
- RFC 4033: DNS Security Introduction and Requirements
 - RFC 4034: Resource Records for the DNS Security Extensions
 - RFC 4035: Protocol Modifications for the DNS Security Extensions
- Related information:**
- IP Security protocols

Berkeley Software Distribution compatibility

Sockets is a Berkeley Software Distribution (BSD) interface.

The semantics, such as the return codes that an application receives and the arguments available on supported functions, are BSD semantics. Some BSD semantics, however, are not available in the IBM i implementation, and changes might need to be made to a typical BSD socket application in order for it to run on the system.

The following list summarizes the differences between the IBM i implementation and the BSD implementation.

/etc/hosts, /etc/services, /etc/networks, and /etc/protocols

For these files, the IBM i implementation supplies the following database files.

QUSRSYS file	Contents
QATOCOST	List of host names and the corresponding IP addresses.
QATOCN	List of networks and the corresponding IP addresses.
QATOCPP	List of protocols that are used in the Internet.
QATOCPS	List of services and the specific port and protocol that the service uses.

/etc/resolv.conf

The IBM i implementation requires that this information is configured using the TCP/IP properties page in IBM Navigator for i. To access the TCP/IP properties page, complete the following steps:

1. From IBM Navigator for i, expand **IBM i Management > Network > All Tasks > TCP/IP Configuration**.
2. Click **TCP/IP Configuration Properties**.

Saving and Restoring TCP/IP configuration information

The Retrieve TCP/IP Information (**RTVTCPINF**) command gathers key TCP/IP configuration information from the default system locations and places it in the library that is specified for the Library (LIB) parameter. The gathered TCP/IP configuration information can be used by the Update TCP/IP Information (**UPDTCPINF**) command to reset or restore the TCP/IP configuration on a system.

bind()

On a BSD system, a client can create an AF_UNIX socket using `socket()`, connect to a server using `connect()`, and then bind a name to its socket using `bind()`. The IBM i implementation does not support this scenario (the `bind()` fails).

close()

The IBM i implementation supports the linger timer for the `close()` API, except for AF_INET sockets over Systems Network Architecture (SNA). Some BSD implementations do not support the linger timer for the `close()` API.

connect()

On a BSD system, if a `connect()` is issued against a socket that was previously connected to an address and is using a connectionless transport service, and an invalid address or an invalid address length is used, the socket is no longer connected. The IBM i implementation does not support this scenario (the `connect()` fails and the socket is still connected).

A connectionless transport socket for which a `connect()` has been issued can be disconnected by setting the `address_length` parameter to zero and issuing another `connect()`.

accept(), getsockname(), getpeername(), recvfrom(), and recvmsg()

When using the AF_UNIX or AF_UNIX_CCSID address family and the socket has not been bound, the default IBM i implementation might return an address length of zero and an unspecified address structure. The IBM i BSD 4.4/ UNIX 98 and other implementations might return a small address structure with only the address family specified.

ioctl()

- On a BSD system, on a socket of type SOCK_DGRAM, the FIONREAD request returns the length of the data plus the length of the address. On the IBM i implementation, FIONREAD only returns the length of data.
- Not all requests available on most BSD implementations of `ioctl()` are available on the IBM i implementation of `ioctl()`.

listen()

On a BSD system, issuing a `listen()` with the backlog parameter set to a value that is less than zero does not result in an error. In addition, the BSD implementation, in some cases, does not use the backlog parameter, or uses an algorithm to come up with a final result for the backlog value. The IBM i implementation returns an error if the backlog value is less than zero. If you set the backlog to a valid value, then the value is used as the backlog. However, setting the backlog to a value larger than {SOMAXCONN}, the backlog defaults to the value set in {SOMAXCONN}.

Out-of-band (OOB) data

In the IBM i implementation, OOB data is not discarded if SO_OOBINLINE is not set, OOB data has been received, and the user then sets SO_OOBINLINE on. The initial OOB byte is considered normal data.

protocol parameter of socket()

As a means of providing additional security, no user is allowed to create a SOCK_RAW socket specifying a protocol of IPPROTO_TCP or IPPROTO_UDP.

res_xlate() and res_close()

These APIs are included in the resolver routines for the IBM i implementation. The `res_xlate()` API translates Domain Name System (DNS) packets from EBCDIC to ASCII and from ASCII to EBCDIC. The `res_close()` API is used to close a socket that was used by the `res_send()` API with the `RES_STAYOPEN` option set. The `res_close()` API also resets the `_res` structure.

sendmsg() and recvmsg()

The IBM i implementation of `sendmsg()` and `recvmsg()` allows `{MSG_MAXIOVLEN}` I/O vectors. The BSD implementation allows `{MSG_MAXIOVLEN - 1}` I/O vectors.

Signals

There are several differences relating to signal support:

- BSD implementations issue a SIGIO signal each time an acknowledgement is received for data sent on an output operation. The IBM i sockets implementation does not generate signals related to outbound data.
- The default action for the SIGPIPE signal is to end the process in BSD implementations. To maintain downward compatibility with previous releases of IBM i, the IBM i implementation uses a default action of ignoring for the SIGPIPE signal.

SO_REUSEADDR option

On BSD systems, a `connect()` call on a socket of family `AF_INET` and type `SOCK_DGRAM` causes the system to change the address to which the socket is bound to the address of the interface that is used to reach the address specified on the `connect()` API. For example, if you bind a socket of type `SOCK_DGRAM` to address `INADDR_ANY`, and then connect it to an address of `a.b.c.d`, the system changes your socket so it is now bound to the IP address of the interface that was chosen to route packets to address `a.b.c.d`. In addition, if this IP address that the socket is bound to is `a.b.c.e`, for example, address `a.b.c.e` now appears on the `getsockname()` API instead of `INADDR_ANY`, and the `SO_REUSEADDR` option must be used to bind any other sockets to the same port number with an address of `a.b.c.e`.

In contrast, in this example, the IBM i implementation does not change the local address from `INADDR_ANY` to `a.b.c.e`. The `getsockname()` API continues to return `INADDR_ANY` after the connection is performed.

SO_SNDBUF and SO_RCVBUF options

The values set for `SO_SNDBUF` and `SO_RCVBUF` on a BSD system provide a greater level of control than on an IBM i implementation. On an IBM i implementation, these values are taken as advisory values.

Related concepts:

“How sockets work” on page 3

Sockets are commonly used for client and server interaction. Typical system configuration places the server on one machine, with the clients on other machines. The clients connect to the server, exchange information, and then disconnect.

Related reference:

“Example: Using signals with blocking socket APIs” on page 164

When a process or an application becomes blocked, signals allow you to be notified. They also provide a time limit for blocking processes.

Related information:

`accept()`--Wait for Connection Request and Make Connection API

--Send a Message Over a Socket API

`connect()`--Establish Connection or Destination Address API

`recvfrom()`--Receive Data API

`recvmsg()`--Receive a Message Over a Socket API

`bind()`--Set Local Address for Socket API

getsockname()--Retrieve Local Address of Socket API
 socket()--Create Socket API
 listen()--Invite Incoming Connections Requests API
 ioctl()--Perform I/O Control Request API
 getpeername()--Retrieve Destination Address of Socket API
 close()--Close File or Socket Descriptor API
 RTVTCPINF--Retrieve TCP/IP Information
 UPDTCPIF--Update TCP/IP Information

UNIX 98 compatibility

Created by The Open Group, a consortium of developers and venders, UNIX 98 improved the inter-operability of the UNIX operating system while incorporating much of the Internet-related function for which UNIX had become known.

IBM i sockets provide programmers the ability to write socket applications that are compatible with UNIX 98 operating environment. Currently, IBM supports two versions of most sockets APIs. The base IBM i socket APIs use Berkeley Socket Distribution (BSD) 4.3 structures and syntax. The other uses syntax and structures compatible with BSD 4.4 and the UNIX 98 programming interface specifications. You can select the UNIX 98 compatible interface by defining the `_XOPEN_SOURCE` macro to a value of 520 or greater.

Differences in address structure for UNIX 98 compatible applications

When you specify the `_XOPEN_OPEN` macro, you can write UNIX 98 compatible applications with the same address families that are used in default IBM i implementations; however, there are differences in the `sockaddr` address structure. The table compares the BSD 4.3 `sockaddr` address structure with the UNIX 98 compatible address structure:

Table 15. Comparison of BSD 4.3 and UNIX 98/BSD 4.4 socket address structure

BSD 4.3 structure	BSD 4.4/ UNIX 98 compatible structure
sockaddr address structure	
<pre>struct sockaddr { u_short sa_family; char sa_data[14]; };</pre>	<pre>struct sockaddr { uint8_t sa_len; sa_family_t sa_family; char sa_data[14]; };</pre>
sockaddr_in address structure	
<pre>struct sockaddr_in { short sin_family; u_short sin_port; struct in_addr sin_addr; char sin_zero[8]; };</pre>	<pre>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]; };</pre>
sockaddr_in6 address structure	
<pre>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; };</pre>	<pre>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; };</pre>

Table 15. Comparison of BSD 4.3 and UNIX 98/BSD 4.4 socket address structure (continued)

BSD 4.3 structure	BSD 4.4/ UNIX 98 compatible structure
sockaddr_un address structure	
<pre>struct sockaddr_un { short sun_family; char sun_path[126]; };</pre>	<pre>struct sockaddr_un { uint8_t sun_len; sa_family_t sun_family; char sun_path[126] };</pre>

API differences

When you develop in ILE-based languages and an application is compiled with the `_XOPEN_SOURCE` macro, some sockets APIs are mapped to internal names. These internal names provide the same function as the original API. The table lists these affected APIs. If you are writing socket applications in some other C-based language, you can write directly to the internal name of these APIs. Use the link to the original API to see usage notes and details for both versions of these APIs.

Table 16. API and UNIX 98 equivalent name

API name	Internal name
<code>accept()</code>	<code>qso_accept98()</code>
<code>accept_and_recv()</code>	<code>qso_accept_and_recv98()</code>
<code>bind()</code>	<code>qso_bind98()</code>
<code>bind2addrsel()</code>	<code>qso_bind2addrsel98()</code>
<code>connect()</code>	<code>qso_connect98()</code>
<code>endhostent()</code>	<code>qso_endhostent98()</code>
<code>endnetent()</code>	<code>qso_endnetent98()</code>
<code>endprotoent()</code>	<code>qso_endprotoent98()</code>
<code>endservent()</code>	<code>qso_endservent98()</code>
<code>getaddrinfo()</code>	<code>qso_getaddrinfo98()</code>
<code>gethostbyaddr()</code>	<code>qso_gethostbyaddr98()</code>
<code>gethostbyaddr_r()</code>	<code>qso_gethostbyaddr_r98()</code>
<code>gethostname()</code>	<code>qso_gethostname98()</code>
<code>gethostname_r()</code>	<code>qso_gethostname_r98()</code>
<code>gethostbyname()</code>	<code>qso_gethostbyname98()</code>
<code>gethostent()</code>	<code>qso_gethostent98()</code>
<code>getnameinfo()</code>	<code>qso_getnameinfo98()</code>
<code>getnetbyaddr()</code>	<code>qso_getnetbyaddr98()</code>
<code>getnetbyname()</code>	<code>qso_getnetbyname98()</code>
<code>getnetent()</code>	<code>qso_getnetent98()</code>
<code>getpeername()</code>	<code>qso_getpeername98()</code>
<code>getprotobyname()</code>	<code>qso_getprotobyname98()</code>
<code>getprotobynumber()</code>	<code>qso_getprotobynumber98()</code>
<code>getprotoent()</code>	<code>qso_getprotoent98()</code>
<code>getsockname()</code>	<code>qso_getsockname98()</code>
<code>getsockopt()</code>	<code>qso_getsockopt98()</code>

Table 16. API and UNIX 98 equivalent name (continued)

API name	Internal name
getservbyname()	qso_getservbyname98()
getservbyport()	qso_getservbyport98()
getservent()	qso_getservent98()
getsourcefilter()	qso_getsourcefilter98()
inet_addr()	qso_inet_addr98()
inet_lnaof()	qso_inet_lnaof98()
inet_makeaddr()	qso_inet_makeaddr98()
inet_netof()	qso_inet_netof98()
inet_network()	qso_inet_network98()
inet6_is_srcaddr()	qso_inet6_is_srcaddr98()
listen()	qso_listen98()
Rbind()	qso_Rbind98()
recv()	qso_recv98()
recvfrom()	qso_recvfrom98()
recvmsg()	qso_recvmsg98()
send()	qso_send98()
sendmsg()	qso_sendmsg98()
sendto()	qso_sendto98()
sethostent()	qso_sethostent98()
setnetent()	qso_setnetent98()
setprotoent()	qso_setprotoent98()
setservent()	qso_setservent98()
setsockopt()	qso_setsockopt98()
setsourcefilter()	qso_setsourcefilter98()
shutdown()	qso_shutdown98()
socket()	qso_socket98()
socketpair()	qso_socketpair98()

Related concepts:

“How sockets work” on page 3

Sockets are commonly used for client and server interaction. Typical system configuration places the server on one machine, with the clients on other machines. The clients connect to the server, exchange information, and then disconnect.

Related information:

accept()--Wait for Connection Request and Make Connection API

accept_and_recv()

connect()--Establish Connection or Destination Address API

--Send a Message Over a Socket API

recvfrom()--Receive Data API

recvmsg()--Receive a Message Over a Socket API

Rbind()--Set Remote Address for Socket API

recv()--Receive Data API

bind()--Set Local Address for Socket API
getsockname()--Retrieve Local Address of Socket API
socket()--Create Socket API
socketpair()--Create a Pair of Sockets API
listen()--Invite Incoming Connections Requests API
ioctl()--Perform I/O Control Request API
getpeername()--Retrieve Destination Address of Socket API
close()--Close File or Socket Descriptor API
endhostent()
endnetent()
endprotoent()
endservent()
gethostbyname()--Get Host Information for Host Name API
getaddrinfo()--Get Address Information API
gethostbyaddr()--Get Host Information for IP Address API
getnameinfo()--Get Name Information for Socket Address API
gethostname()
gethostent()
getnetbyaddr()
getnetbyname()
getnetent()
getprotobyname()
getprotobynumber()
getprotoent()
getsockopt()
getservbyname()
getservbyport()
getservent()
inet_addr()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
send()--Send Data API
sendto()--Send Data API
sethostent()
setnetent()
setprotoent()
setservent()
setsockopt()--Set Socket Options API

Descriptor passing between processes: sendmsg() and recvmsg()

Passing an open descriptor between jobs allows one process (typically a server) to do everything that is required to obtain the descriptor, such as opening a file, establishing a connection, and waiting for the accept() API to complete. It also allows another process (typically a worker) to handle all the data transfer operations as soon as the descriptor is open.

The ability to pass an open descriptor between jobs can lead to a new way of designing client/server applications. This design results in simpler logic for both the server and the worker jobs. This design also allows different types of worker jobs to be easily supported. The server can make a simple check to determine which type of worker should receive the descriptor.

Sockets provide three sets of APIs that can pass descriptors between server jobs:

- `spawn()`

Note: `spawn()` is not a socket API. It is supplied as part of the IBM i Process-Related APIs.

- `givedescriptor()` and `takedescriptor()`
- `sendmsg()` and `recvmsg()`

The `spawn()` API starts a new server job (often called a "child job") and gives certain descriptors to that child job. If the child job is already active, then the `givedescriptor()` and `takedescriptor()` or the `sendmsg()` and `recvmsg()` APIs need to be used.

However, the `sendmsg()` and `recvmsg()` APIs offer many advantages over `spawn()` and `givedescriptor()` and `takedescriptor()`:

Portability

The `givedescriptor()` and `takedescriptor()` APIs are nonstandard and unique to the IBM i operating system. If the portability of an application between the IBM i operating system and UNIX is an issue, you might want to use the `sendmsg()` and `recvmsg()` APIs instead.

Communication of control information

Often the worker job needs to know additional information when it receives a descriptor, such as:

- What type of descriptor is it?
- What should the worker job do with it?

The `sendmsg()` and `recvmsg()` APIs allow you to transfer data, which might be control information, along with the descriptor; the `givedescriptor()` and `takedescriptor()` APIs do not.

Performance

Applications that use the `sendmsg()` and `recvmsg()` APIs tend to perform slightly better than those that use the `givedescriptor()` and `takedescriptor()` APIs in three areas:

- Elapsed time
- CPU utilization
- Scalability

The amount of performance improvement of an application depends on the extent that the application passes descriptors.

Pool of worker jobs

You might want to set up a pool of worker jobs so that a server can pass a descriptor and only one of the jobs in the pool becomes active and receives the descriptor. The `sendmsg()` and `recvmsg()` APIs can be used to accomplish this by having all of the worker jobs wait on a shared descriptor. When the server calls `sendmsg()`, only one of the worker jobs receives the descriptor.

Unknown worker job ID

The `givedescriptor()` API requires the server job to know the job identifier of the worker job. Typically the worker job obtains the job identifier and transfers it over to the server job with a data queue. The `sendmsg()` and `recvmsg()` do not require the extra overhead to create and manage this data queue.

Adaptive server design

When a server is designed using the `givedescriptor()` and `takedescriptor()`, a data queue is typically used to transfer the job identifiers from worker jobs over to the server. The server then does a `socket()`, `bind()`, `listen()`, and an `accept()`. When the `accept()` API is completed, the server

pulls off the next available job ID from the data queue. It then passes the inbound connection to that worker job. Problems arise when many incoming connection requests occur at once and there are not enough worker jobs available. If the data queue that contains the worker job identifiers is empty, the server blocks waiting for a worker job to become available, or the server creates additional worker jobs. In many environments, neither of these alternatives are what you want because additional incoming requests might fill the listen backlog.

Servers that use `sendmsg()` and `recvmsg()` APIs to pass descriptors remain unhindered during heavy activity because they do not need to know which worker job is going to handle each incoming connection. When a server calls `sendmsg()`, the descriptor for the incoming connection and any control data are put into an internal queue for the `AF_UNIX` socket. When a worker job becomes available, it calls `recvmsg()` and receives the first descriptor and the control data that was in the queue.

Inactive worker job

The `givedescriptor()` API requires the worker job to be active while the `sendmsg()` API does not. The job that calls `sendmsg()` does not require any information about the worker job. The `sendmsg()` API requires only that an `AF_UNIX` socket connection has been set up.

An example of how the `sendmsg()` API can be used to pass a descriptor to a job that does not exist follows:

A server can use the `socketpair()` API to create a pair of `AF_UNIX` sockets, use the `sendmsg()` API to send a descriptor over one of the `AF_UNIX` sockets created by `socketpair()`, and then call `spawn()` to create a child job that inherits the other end of the socket pair. The child job calls `recvmsg()` to receive the descriptor that the server passed. The child job was not active when the server called `sendmsg()`.

Pass more than one descriptor at a time

The `givedescriptor()` and `takedescriptor()` APIs allow only one descriptor to be passed at a time. The `sendmsg()` and `recvmsg()` APIs can be used to pass an array of descriptors.

Related reference:

“Example: Passing descriptors between processes” on page 101

These examples demonstrate how to design a server program using the `sendmsg()` and `recvmsg()` APIs to handle incoming connections.

Related information:

`socketpair()`--Create a Pair of Sockets API

Sockets-related User Exit Points

Sockets-related user exit points give an exit program the ability to prevent a specific sockets API from completing successfully.

Sockets-related user exit points give an exit program the ability to control connections based on specific conditions for a job at runtime. This functionality is provided through system-wide user exit points for sockets APIs accepting incoming connections, `connect()`, and `listen()`. The user exit can allow or deny the operation successful completion based on the criteria set by the registered exit program. The intent is to allow exit programs runtime determination if a particular operation is allowed to complete based on the characteristics of the requesting job. These characteristics can include things such as user ID, job type, time of day, current system usage, and so on.

Exit points defined in the User Registry

User-defined exit programs registered with the exit points defined in the user registry are able to limit incoming and outgoing connections. The return codes of the user-defined exit programs indicate whether to allow successful completion to `connect()`, `listen()`, `accept()`, `accept_and_recv()`, or `QsoStartAccept()`.

Table 17. Sockets-related User Exit Points

User Exit Point	Description
QIBM_QSO_ACCEPT	Enables a custom exit program to allow or deny incoming connections based on the restrictions set by the programs.
QIBM_QSO_CONNECT	Enables a custom exit program to allow or deny outgoing connections based on the restrictions set by the programs.
QIBM_QSO_LISTEN	Enables a custom exit program to allow or deny a socket the ability to listen for connections based on the restrictions set by the programs.

Notes:

1. By default, the sockets APIs accepting connections silently ignore rejected connections and wait for the next incoming connection. To give an application the ability to be informed about rejected connections, a socket option is provided. The socket option is enabled by setsockopt() with a level of SOL_SOCKET and option name SO_ACCEPTPERM. When the socket option is enabled, sockets APIs accepting connections fail with EPERM for each incoming connection rejected by the user exit program registered for QIBM_QSO_ACCEPT.
2. Any user trying to add or remove a sockets-related user exit program is required to have *IOSYSCFG, *ALLOBJ, and *SECADM authority.
3. Not all IBM developed applications call the configured user exit programs for one of the following reasons:
 - The application does not use sockets APIs for network communication.
 - The sockets API was called from a system task that is unable to call user exit programs.

Related information:

Sockets accept API Exit Program

Sockets connect() API Exit Program

Sockets listen() API Exit Program

Example: User Exit Program for QIBM_QSO_ACCEPT

An example application to be registered for user exit point QIBM_QSO_ACCEPT. It rejects all incoming connections to the server listening on port 12345 coming from a particular remote IP address between the hours of 12 A.M. and 4 A.M.

This system-wide exit program determines if the incoming connection is allowed to be accepted by the socket API accepting connections or rejected.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Sample User Exit Program for QIBM_QSO_ACCEPT          */
/*                                                         */
/* Exit Point Name : QIBM_QSO_ACCEPT                      */
/*                                                         */
/* Description      : The following ILE C language program */
/*                   will reject all incoming connections to */
/*                   the server listening on port 12345 coming */
/*                   from the remote IP address of '192.0.2.1' or */
/*                   '2001:DB8::1' between the hours of 12 A.M. */
/*                   and 4 A.M.                             */
/*                                                         */
*****/
#include <stdio.h>
#include <string.h>

```

```

#include <esoextpt.h>                                /* Exit program formats */
#include <netinet/in.h>
#include <arpa/inet.h>
int main(int argc, char *argv[])
{
    Qso_ACPT0100_Format_t input;                    /* input format */
    struct in_addr addr4;
    struct in6_addr addr6;
    void *addr, *compareaddr;
    char return_code;
    int comparelen, port;
    /******
    /* Initialize the address to compare 192.0.2.1 and 2001:DB8::1 */
    /******
    inet_pton(AF_INET, "192.0.2.1", &addr4);
    inet_pton(AF_INET6, "2001:DB8::1", &addr6);

    /******
    /* By default allow the connection. */
    /******
    return_code = '0';

    /******
    /* Copy format parameter to local storage. */
    /******
    memcpy(&input, (Qso_ACPT0100_Format_t *) argv[1],
           sizeof(Qso_ACPT0100_Format_t));

    /******
    /* Determine if we have an IPv4 or IPv6 address */
    /******
    if(input.Local_Incoming_Address_Length ==
        sizeof(struct sockaddr_in))
    {
        compareaddr = &addr4;
        comparelen = sizeof(addr4);
        addr = &input.Remote_Address.sinstruct.sin_addr;
        port = input.Local_Incoming_Address.sinstruct.sin_port;
    }
    else
    {
        compareaddr = &addr6;
        comparelen = sizeof(addr6);
        addr = &input.Remote_Address.sin6struct.sin6_addr;
        port = input.Local_Incoming_Address.sin6struct.sin6_port;
    }

    /******
    /* If the local port is 12345 and the incoming connection is */
    /* from 192.0.2.1 or 2001:DB8::1 */
    /******
    if(port == 12345 && (memcmp(addr, compareaddr, comparelen) == 0))
    {
        /******
        /* And the time is between 12 A.M. and 4 A.M. */
        /* Reject the connection. */
        /******
        if(IsTimeBetweenMidnightAnd4AM())
            return_code = '1';
    }
    *argv[2] = return_code;
    return 0;
}

```

Socket scenario: Creating an application to accept IPv4 and IPv6 clients

This example shows a typical situation in which you might want to use the AF_INET6 address family.

Situation

Suppose that you are a socket programmer who works for an application development company that specializes in socket applications for the IBM i operating system. To keep ahead of its competitors, your company has decided to develop a suite of applications that use the AF_INET6 address family, which accept connections from IPv4 and IPv6. You want to create an application that processes requests from both IPv4 and IPv6 nodes. You know that the IBM i operating system supports the AF_INET6 address family sockets, which provides interoperability with AF_INET address family sockets. You also know that you can accomplish this by using an IPv4-mapped IPv6 address format.

Scenario objectives

This scenario has the following objectives and goals:

1. Create a server application that accepts and processes requests from IPv6 and IPv4 clients
2. Create a client application that requests data from an IPv4 or IPv6 server application

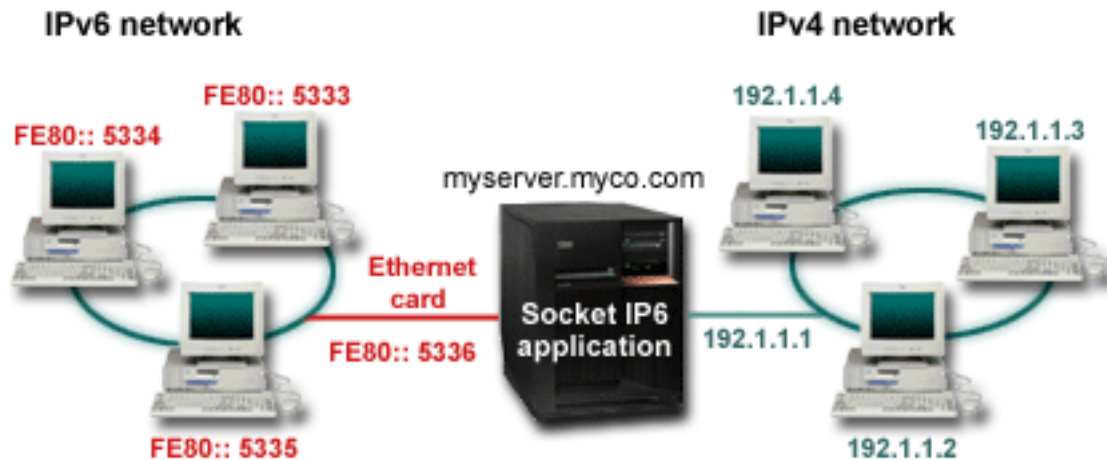
Prerequisite steps

Before developing your application that meets these objectives, complete the following tasks:

1. Install QSYSINC library. This library provides necessary header files that are needed when compiling socket applications.
2. Install the ILE C licensed program (5770-WDS option 51).
3. Install and configure an Ethernet card. For information about Ethernet options, see the Ethernet topic in the information center.
4. Set up TCP/IP and IPv6 network. Refer to the information about configuring TCP/IP and configuring IPv6.

Scenario details

The following graphic describes the IPv6 network, for which you create applications to handle requests from IPv6 and IPv4 clients. The IBM i operating system contains the program that listens and processes requests from these clients. The network consists of two separate domains, one that contains IPv4 clients exclusively and the other remote network that contains only IPv6 clients. The domain name of the system is myserver.myco.com. The server application uses the AF_INET6 address family to process these incoming requests with the `in6addr_any` specified on the `bind()` API call.



Related reference:

“Using AF_INET6 address family” on page 28

AF_INET6 sockets provide support for Internet Protocol version 6 (IPv6) 128 bit (16 byte) address structures. Programmers can write applications using the AF_INET6 address family to accept client requests from either IPv4 or IPv6 nodes, or from IPv6 nodes only.

Related information:

Ethernet

Configuring TCP/IP for the first time

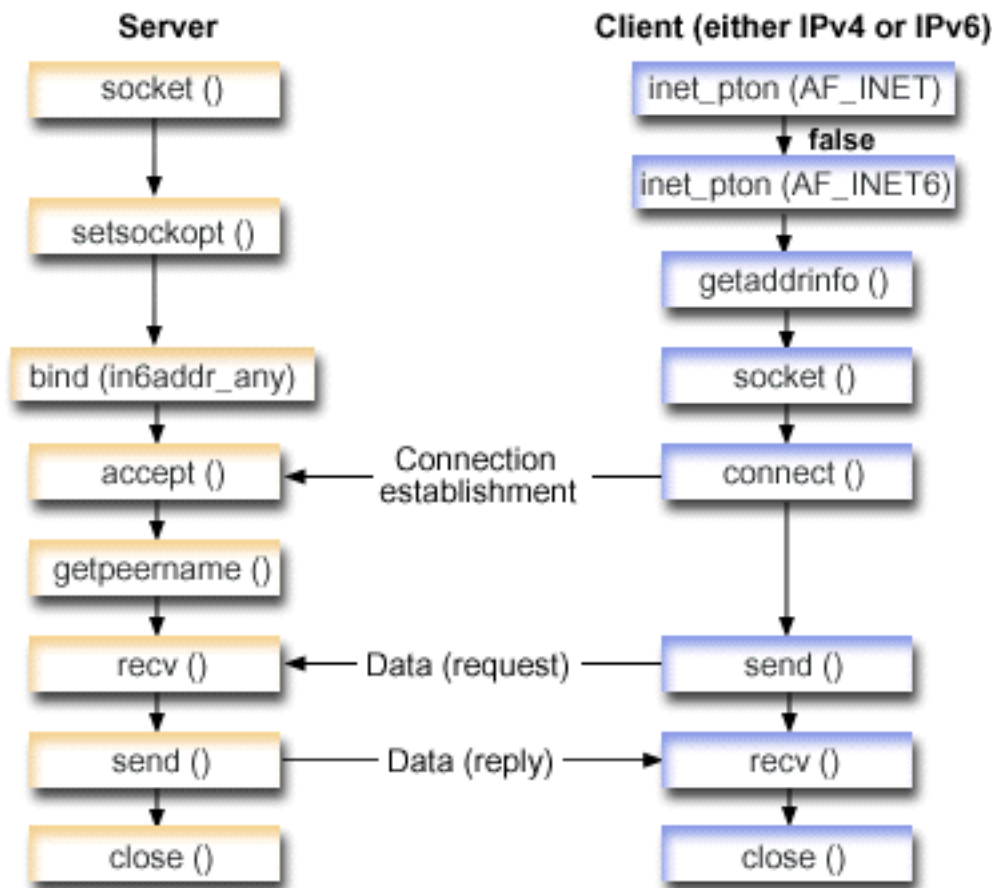
Configuring IPv6

Example: Accepting connections from both IPv6 and IPv4 clients

This example program demonstrates how to create a server/client model that accepts requests from both IPv4 (those socket applications that use the AF_INET address family) and IPv6 (those applications that use the AF_INET6 address family).

Currently your socket application can only use the AF_INET address family, which allows for TCP and User Datagram Protocol (UDP) protocol; however, this might change with the increase in the use of IPv6 addresses. You can use this sample program to create your own applications that accommodate both address families.

This figure shows how this example program works:



Socket flow of events: Server application that accepts requests from both IPv4 and IPv6 clients

This flow describes each of the API calls and what they do within the socket application that accepts requests from both IPv4 and IPv6 clients.

1. The `socket()` API specifies a socket descriptor that creates an endpoint. It also specifies the `AF_INET6` address family, which supports IPv6, and the TCP transport (`SOCK_STREAM`) is used for this socket.
2. The `setsockopt()` API allows an application to reuse the local address when the server is restarted before the required wait time expires.
3. A `bind()` API supplies a unique name for the socket. In this example, the programmer sets the address to `in6addr_any`, which (by default) allows connections to be established from any IPv4 or IPv6 client that specifies port 3005 (that is, the bind is done to both the IPv4 and IPv6 port spaces).

Note: If the server only needs to handle IPv6 clients, then `IPv6_ONLY` socket option can be used.

4. The `listen()` API allows the server to accept incoming client connections. In this example, the programmer sets the backlog to 10, which allows the system to queue ten connection requests before the system starts rejecting incoming requests.
5. The server uses the `accept()` API to accept an incoming connection request. The `accept()` call blocks indefinitely, waiting for the incoming connection to arrive from an IPv4 or IPv6 client.
6. The `getpeername()` API returns the client's address to the application. If the client is an IPv4 client, the address is shown as an IPv4-mapped IPv6 address.

7. The `recv()` API receives 250 bytes of data from the client. In this example, the client sends 250 bytes of data over. Knowing this, the programmer uses the `SO_RCVLOWAT` socket option and specifies that the `recv()` API to not wake up until all 250 bytes of data have arrived.
8. The `send()` API echoes the data back to the client.
9. The `close()` API closes any open socket descriptors.

Socket flow of events: Requests from either IPv4 or IPv6 clients

Note: This client example can be used with other server application designs that want to accept request for either IPv4 or IPv6 nodes. Other server designs can be used with this client example.

1. The `inet_pton()` call converts the text form of the address to the binary form. In this example, two of these calls are issued. The first determines if the server is a valid `AF_INET` address. The second `inet_pton()` call determines whether the server has an `AF_INET6` address. If it is numeric, `getaddrinfo()` should be prevented from doing any name resolution. Otherwise a host name was provided that needs to be resolved when the `getaddrinfo()` call is issued.
2. The `getaddrinfo()` call retrieves the address information needed for the subsequent `socket()` and `connect()` calls.
3. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies the address family, socket type, and protocol using the information returned from the `getaddrinfo()` API call.
4. The `connect()` API establishes a connection to the server regardless of whether the server is IPv4 or IPv6.
5. The `send()` API sends the data request to the server.
6. The `recv()` API receives data from the server application.
7. The `close()` API closes any open socket descriptors.

The following sample code shows the server application for this scenario.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Header files needed for this sample program */
*****/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*****
/* Constants used by this program */
*****/
#define SERVER_PORT    3005
#define BUFFER_LENGTH  250
#define FALSE          0

void main()
{
    /*****
    /* Variable and structure definitions. */
    *****/
    int sd=-1, sdconn=-1;
    int rc, on=1, rcdsize=BUFFER_LENGTH;
    char buffer[BUFFER_LENGTH];
    struct sockaddr_in6 serveraddr, clientaddr;
    int addrlen=sizeof(clientaddr);
    char str[INET6_ADDRSTRLEN];

```



```

/*****/
/* A do/while(FALSE) loop is used to make error cleanup easier. The */
/* close() of each of the socket descriptors is only done once at the */
/* very end of the program. */
/*****/
do
{
    /*****/
    /* The socket() function returns a socket descriptor, which represents */
    /* an endpoint. Get a socket for address family AF_INET6 to */
    /* prepare to accept incoming connections on. */
    /*****/
    if ((sd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
    {
        perror("socket() failed");
        break;
    }

    /*****/
    /* The setsockopt() function is used to allow the local address to */
    /* be reused when the server is restarted before the required wait */
    /* time expires. */
    /*****/
    if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
        (char *)&on, sizeof(on)) < 0)
    {
        perror("setsockopt(SO_REUSEADDR) failed");
        break;
    }

    /*****/
    /* After the socket descriptor is created, a bind() function gets a */
    /* unique name for the socket. In this example, the user sets the */
    /* address to in6addr_any, which (by default) allows connections to */
    /* be established from any IPv4 or IPv6 client that specifies port */
    /* 3005. (that is, the bind is done to both the IPv4 and IPv6 TCP/IP */
    /* stacks). This behavior can be modified using the IPPROTO_IPV6 */
    /* level socket option IPV6_V6ONLY if required. */
    /*****/
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_port = htons(SERVER_PORT);
    /*****/
    /* Note: applications use in6addr_any similarly to the way they use */
    /* INADDR_ANY in IPv4. A symbolic constant IN6ADDR_ANY_INIT also */
    /* exists but can only be used to initialize an in6_addr structure */
    /* at declaration time (not during an assignment). */
    /*****/
    serveraddr.sin6_addr = in6addr_any;
    /*****/
    /* Note: the remaining fields in the sockaddr_in6 are currently not */
    /* supported and should be set to 0 to ensure upward compatibility. */
    /*****/

    if (bind(sd,
        (struct sockaddr *)&serveraddr,
        sizeof(serveraddr)) < 0)
    {
        perror("bind() failed");
        break;
    }

    /*****/
    /* The listen() function allows the server to accept incoming */
    /* client connections. In this example, the backlog is set to 10. */
    /* This means that the system will queue 10 incoming connection */
    /*

```

```

/* requests before the system starts rejecting the incoming */
/* requests. */
/*****
if (listen(sd, 10) < 0)
{
    perror("listen() failed");
    break;
}

printf("Ready for client connect().\n");

/*****
/* The server uses the accept() function to accept an incoming */
/* connection request. The accept() call will block indefinitely */
/* waiting for the incoming connection to arrive from an IPv4 or */
/* IPv6 client. */
/*****
if ((sdconn = accept(sd, NULL, NULL)) < 0)
{
    perror("accept() failed");
    break;
}
else
{
    /*****
    /* Display the client address. Note that if the client is */
    /* an IPv4 client, the address will be shown as an IPv4 Mapped */
    /* IPv6 address. */
    /*****
    getpeername(sdconn, (struct sockaddr *)&clientaddr, &addrlen);
    if(inet_ntop(AF_INET6, &clientaddr.sin6_addr, str, sizeof(str))) {
        printf("Client address is %s\n", str);
        printf("Client port is %d\n", ntohs(clientaddr.sin6_port));
    }
}

/*****
/* In this example we know that the client will send 250 bytes of */
/* data over. Knowing this, we can use the SO_RCVLOWAT socket */
/* option and specify that we don't want our recv() to wake up */
/* until all 250 bytes of data have arrived. */
/*****
if (setsockopt(sdconn, SOL_SOCKET, SO_RCVLOWAT,
    (char *)&rsize, sizeof(rsize)) < 0)
{
    perror("setsockopt(SO_RCVLOWAT) failed");
    break;
}

/*****
/* Receive that 250 bytes of data from the client */
/*****
rc = recv(sdconn, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("recv() failed");
    break;
}

printf("%d bytes of data were received\n", rc);
if (rc == 0 ||
    rc < sizeof(buffer))
{
    printf("The client closed the connection before all of the\n");
    printf("data was sent\n");
    break;
}

```

```

/*****
/* Echo the data back to the client */
/*****
rc = send(sdconn, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("send() failed");
    break;
}

/*****
/* Program complete */
/*****

} while (FALSE);

/*****
/* Close down any open socket descriptors */
/*****
if (sd != -1)
    close(sd);
if (sdconn != -1)
    close(sdconn);
}

```

Related reference:

“Examples: Connection-oriented designs” on page 91

You can design a connection-oriented socket server on the system in a number of ways. These example programs can be used to create your own connection-oriented designs.

“Example: IPv4 or IPv6 client”

This sample program can be used with the server application that accepts requests from either IPv4 or IPv6 clients.

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a socket(), connect(), send(), recv(), and close() operation.

Related information:

socket()--Create Socket API

setsockopt()--Set Socket Options API

bind()--Set Local Address for Socket API

listen()--Invite Incoming Connections Requests API

accept()--Wait for Connection Request and Make Connection API

getpeername()--Retrieve Destination Address of Socket API

recv()--Receive Data API

send()--Send Data API

close()--Close File or Socket Descriptor API

inet_pton()

getaddrinfo()--Get Address Information API

connect()--Establish Connection or Destination Address API

Example: IPv4 or IPv6 client

This sample program can be used with the server application that accepts requests from either IPv4 or IPv6 clients.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This is an IPv4 or IPv6 client.
*****/

/*****
/* Header files needed for this sample program
*****/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*****
/* Constants used by this program
*****/
#define BUFFER_LENGTH    250
#define FALSE            0
#define SERVER_NAME      "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or */
/* set the server name in the #define      */
/* SERVER_NAME.                            */
void main(int argc, char *argv[])
{
    /*****
    /* Variable and structure definitions.
    *****/
    int     sd=-1, rc, bytesReceived=0;
    char    buffer[BUFFER_LENGTH];
    char    server[NETDB_MAX_HOST_NAME_LENGTH];
    char    servport[] = "3005";
    struct  in6_addr serveraddr;
    struct  addrinfo hints, *res=NULL;

    /*****
    /* A do/while(FALSE) loop is used to make error cleanup easier. The */
    /* close() of the socket descriptor is only done once at the very end */
    /* of the program along with the free of the list of addresses.
    *****/
    do
    {
        /*****
        /* If an argument was passed in, use this as the server, otherwise */
        /* use the #define that is located at the top of this program.
        *****/
        if (argc > 1)
            strcpy(server, argv[1]);
        else
            strcpy(server, SERVER_NAME);

        memset(&hints, 0x00, sizeof(hints));
        hints.ai_flags    = AI_NUMERICSERV;
        hints.ai_family   = AF_UNSPEC;
        hints.ai_socktype = SOCK_STREAM;

        /*****
        /* Check if we were provided the address of the server using */
        /* inet_pton() to convert the text form of the address to binary */
        /* form. If it is numeric then we want to prevent getaddrinfo() */
        /* from doing any name resolution.
        *****/
        rc = inet_pton(AF_INET, server, &serveraddr);
        if (rc == 1)    /* valid IPv4 text address? */
        {

```

```

    hints.ai_family = AF_INET;
    hints.ai_flags |= AI_NUMERICHOST;
}
else
{
    rc = inet_pton(AF_INET6, server, &serveraddr);
    if (rc == 1) /* valid IPv6 text address? */
    {
        hints.ai_family = AF_INET6;
        hints.ai_flags |= AI_NUMERICHOST;
    }
}
/*****/
/* Get the address information for the server using getaddrinfo(). */
/*****/
rc = getaddrinfo(server, servport, &hints, &res);
if (rc != 0)
{
    printf("Host not found --> %s\n", gai_strerror(rc));
    if (rc == EAI_SYSTEM)
        perror("getaddrinfo() failed");
    break;
}

/*****/
/* The socket() function returns a socket descriptor, which represents */
/* an endpoint. The statement also identifies the address family, */
/* socket type, and protocol using the information returned from */
/* getaddrinfo(). */
/*****/
sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (sd < 0)
{
    perror("socket() failed");
    break;
}

/*****/
/* Use the connect() function to establish a connection to the */
/* server. */
/*****/
rc = connect(sd, res->ai_addr, res->ai_addrlen);
if (rc < 0)
{
    /*****/
    /* Note: the res is a linked list of addresses found for server. */
    /* If the connect() fails to the first one, subsequent addresses */
    /* (if any) in the list can be tried if required. */
    /*****/
    perror("connect() failed");
    break;
}

/*****/
/* Send 250 bytes of a's to the server */
/*****/
memset(buffer, 'a', sizeof(buffer));
rc = send(sd, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("send() failed");
    break;
}

/*****/
/* In this example we know that the server is going to respond with */
/* the same 250 bytes that we just sent. Since we know that 250 */

```

```

/* bytes are going to be sent back to us, we can use the */
/* SO_RCVLOWAT socket option and then issue a single recv() and */
/* retrieve all of the data. */
/*
/* The use of SO_RCVLOWAT is already illustrated in the server */
/* side of this example, so we will do something different here. */
/* The 250 bytes of the data may arrive in separate packets, */
/* therefore we will issue recv() over and over again until all */
/* 250 bytes have arrived. */
/*****/
while (bytesReceived < BUFFER_LENGTH)
{
    rc = recv(sd, & buffer[bytesReceived],
              BUFFER_LENGTH - bytesReceived, 0);
    if (rc < 0)
    {
        perror("recv() failed");
        break;
    }
    else if (rc == 0)
    {
        printf("The server closed the connection\n");
        break;
    }

    /*****/
    /* Increment the number of bytes that have been received so far */
    /*****/
    bytesReceived += rc;
}

} while (FALSE);

/*****/
/* Close down any open socket descriptors */
/*****/
if (sd != -1)
    close(sd);
/*****/
/* Free any results returned from getaddrinfo */
/*****/
if (res != NULL)
    freeaddrinfo(res);
}

```

Related reference:

“Example: Accepting connections from both IPv6 and IPv4 clients” on page 80

This example program demonstrates how to create a server/client model that accepts requests from both IPv4 (those socket applications that use the AF_INET address family) and IPv6 (those applications that use the AF_INET6 address family).

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a socket(), connect(), send(), recv(), and close() operation.

Socket application design recommendations

Before working with a socket application, assess the **functional requirements**, **goals**, and **needs** of the socket application. Also, consider the **performance** requirements and the **system resource** impacts of the application.

The following list of recommendations helps you address some of these issues for your socket application and points out **better ways to use sockets** and to design your socket applications:

Table 18. Socket design applications

Recommendation	Reason	Best used in
Use asynchronous I/O	Asynchronous I/O used in a threaded server model is preferable over the more conventional select() model.	Socket server applications which handle numerous concurrent clients.
When using asynchronous I/O, adjust the number of threads in the process to an optimum number for the number of clients to be processed .	If too few threads are defined then some clients might time out before being handled. If too many threads are defined then some system resource are not used efficiently. Note: It is better to have too many threads than too few threads.	Socket applications using asynchronous I/O.
Design socket application to avoid the use of the postflag on all start operations for asynchronous I/O.	Avoids the performance overhead of transition to a completion port if the operation has already been satisfied synchronously.	Socket applications using asynchronous I/O.
Use send() and recv() over read() and write().	send() and recv() APIs provide a small performance and serviceability improvement over read() and write().	Any socket program that knows it uses a socket descriptor and not a file descriptor.
Use the receive low water (SO_RCVLOWAT) socket option to avoid looping on a receive operation until all data has arrived.	Allows your application to wait for a minimum amount of data to be received on the socket before satisfying a blocked receive operation.	Any socket application that receives data
Use the MSG_WAITALL flag to avoid looping on a receive operation until all data has arrived.	Allows your application to wait for the entire buffer provided on the receive operation to be received before satisfying a blocked receive operation.	Any socket application that receives data and knows in advance how much it expects to arrive.
Use sendmsg() and recvmsg() over givedescriptor() and takedescriptor().	See "Descriptor passing between processes: sendmsg() and recvmsg()" on page 74 for the advantages.	Any socket application passing socket or file descriptors between processes.
Use poll() over select().	See "Using poll() instead of select()" on page 158 for the advantages.	Any socket application using select() to poll the status of a socket.
When using select(), try to avoid a large number of descriptors in the read, write or exception set. Note: If you have a large number of descriptors being used for select() processing see the asynchronous I/O recommendation above.	When there are a large number of descriptors in a read, write or exception set, considerable redundant work occurs each time select() is called. As soon as a select() is satisfied the actual socket function must still be done, that is, a read or write or accept must still be performed. Asynchronous I/O APIs combine the notification that something has occurred on a socket with the actual I/O operation.	Applications that have a large(> 50) number of descriptors active for select().
Save your copy of the read, write and exception sets before using select() to avoid rebuilding the sets for every time you must reissue the select().	This saves the overhead of rebuilding the read, write, or exception sets every time you plan to issue the select().	Any socket application where you are using select() with a large number of socket descriptors enabled for read, write or exception processing.

Table 18. Socket design applications (continued)

Recommendation	Reason	Best used in
Do not use select() as a timer. Use sleep() instead. Note: If granularity of the sleep() timer is not adequate, you might need to use select() as a timer. In this case, set maximum descriptor to 0 and the read, write, and exception set to NULL.	Better timer response and less system overhead.	Any socket application where you are using select() just as a timer.
If your socket application has increased the maximum number of file and socket descriptors allowed per process using DosSetRelMaxFH() and you are using select() in this same application, be careful of the affect this new maximum value has on the size of the read, write and exception sets used for select() processing.	If you allocate a descriptor outside the range of the read, write or exception set, as specified by FD_SETSIZE, then you can overwrite and destroy storage. Ensure your set sizes are at least large enough to handle whatever the maximum number of descriptors are set for the process and the maximum descriptor value specified on the select() API.	Any application or process where you use DosSetRelMaxFH() and select().
Set all socket descriptors in the read or write sets to nonblocking. When a descriptor becomes enabled for read or write, loop and consume or send all of the data until EWOULDBLOCK is returned.	This allows you to minimize the number of select() calls when data is still available to be processed or read on a descriptor.	Any socket application where you are using select().
Only specify the sets that you need to use for select() processing.	Most applications do not need to specify the exception set or write set.	Any socket application where you are using select().
Use GSKit APIs instead of SSL_APIs.	Both the Global Security Kit (GSKit) and SSL_ APIs allow you to develop secure AF_INET or AF_INET6, SOCK_STREAM socket applications. Because the GSKit APIs are supported across IBM systems, they are the preferred APIs to secure an application. The SSL_ APIs exist only in the IBM i operating system.	Any socket application that needs to be enabled for SSL or TLS processing.
Avoid using signals.	The performance overhead of signals (on all platforms, not just the IBM i platform) is expensive. It is better to design your socket application to use Asynchronous I/O or select() APIs.	Any socket application that uses signals.
Use protocol independent routines when available, such as inet_ntop(), inet_pton(), getaddrinfo(), and getnameinfo().	Even if you are not yet ready to support IPv6, use these APIs, (instead of inet_ntoa(), inet_addr(), gethostbyname() and gethostbyaddr()) to prepare you for easier migration.	Any AF_INET or AF_INET6 application that uses network routines.
Use sockaddr_storage to declare storage for any address family address.	Simplifies writing code portable across multiple address families and platforms. Declares enough storage to hold the largest address family and ensures the correct boundary alignment.	Any socket application that stores addresses.

Related concepts:

“Asynchronous I/O” on page 43

Asynchronous I/O APIs provide a method for threaded client/server models to perform highly concurrent and memory-efficient I/O.

Related reference:

“Example: Using asynchronous I/O” on page 116

An application creates an I/O completion port using the `QsoCreateIOCompletionPort()` API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests.

“Example: Nonblocking I/O and `select()`” on page 152

This sample program illustrates a server application that uses nonblocking and the `select()` API.

“Example: Passing descriptors between processes” on page 101

These examples demonstrate how to design a server program using the `sendmsg()` and `recvmsg()` APIs to handle incoming connections.

Related information:

`DosSetRelMaxFH()`

Examples: Socket application designs

These example programs illustrate the more advanced socket concepts. You can use these example programs to create your own applications that complete a similar task.

With these examples, there are graphics and a listing of calls that illustrate the flow of events in each of these applications. You can use the Xsockets tool interactively, try some of these APIs in these programs, or you can make specific changes for your particular environment.

Examples: Connection-oriented designs

You can design a connection-oriented socket server on the system in a number of ways. These example programs can be used to create your own connection-oriented designs.

While additional socket server designs are possible, the designs provided in these examples are the most common.

Iterative server

In the iterative server example, a single server job handles all incoming connections and all data flows with the client jobs. When the `accept()` API is completed, the server handles the entire transaction. This is the easiest server to develop, but it does have a few problems. While the server is handling the request from a given client, additional clients can be trying to get to the server. These requests fill the `listen()` backlog and some of the them are rejected eventually.

Concurrent server

In the concurrent server designs, the system uses multiple jobs and threads to handle the incoming connection requests. With a concurrent server there are typically multiple clients that connect to the server at the same time.

For multiple concurrent clients in a network, it is recommended that you use the asynchronous I/O socket APIs. These APIs provide the best performance in networks that have multiple concurrent clients.

- `spawn()` server and `spawn()` worker

The `spawn()` API is used to create a new job to handle each incoming request. After `spawn()` is completed, the server can wait on the `accept()` API for the next incoming connection to be received.

The only problem with this server design is the performance overhead of creating a new job each time a connection is received. You can avoid the performance overhead of the `spawn()` server example by

using prestarted jobs. Instead of creating a new job each time a connection is received, the incoming connection is given to a job that is already active. All of the remaining examples in this topic use prestarted jobs.

- `sendmsg()` server and `recvmsg()` worker

The `sendmsg()` and `recvmsg()` APIs are used to handle incoming connections. The server prestarts all of the worker jobs when the server job first starts.

- Multiple `accept()` servers and multiple `accept()` workers

For the previous APIs, the worker job does not get involved until after the server receives the incoming connection request. When the multiple `accept()` APIs are used, each of the worker jobs can be turned into an iterative server. The server job still calls the `socket()`, `bind()`, and `listen()` APIs. When the `listen()` call is completed, the server creates each of the worker jobs and gives a listening socket to each one of them. All of the worker jobs then call the `accept()` API. When a client tries to connect to the server, only one `accept()` call is completed, and that worker handles the connection.

Related concepts:

“Asynchronous I/O” on page 43

Asynchronous I/O APIs provide a method for threaded client/server models to perform highly concurrent and memory-efficient I/O.

Related reference:

“Example: Accepting connections from both IPv6 and IPv4 clients” on page 80

This example program demonstrates how to create a server/client model that accepts requests from both IPv4 (those socket applications that use the `AF_INET` address family) and IPv6 (those applications that use the `AF_INET6` address family).

“Example: Using asynchronous I/O” on page 116

An application creates an I/O completion port using the `QsoCreateIOCompletionPort()` API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests.

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()` operation.

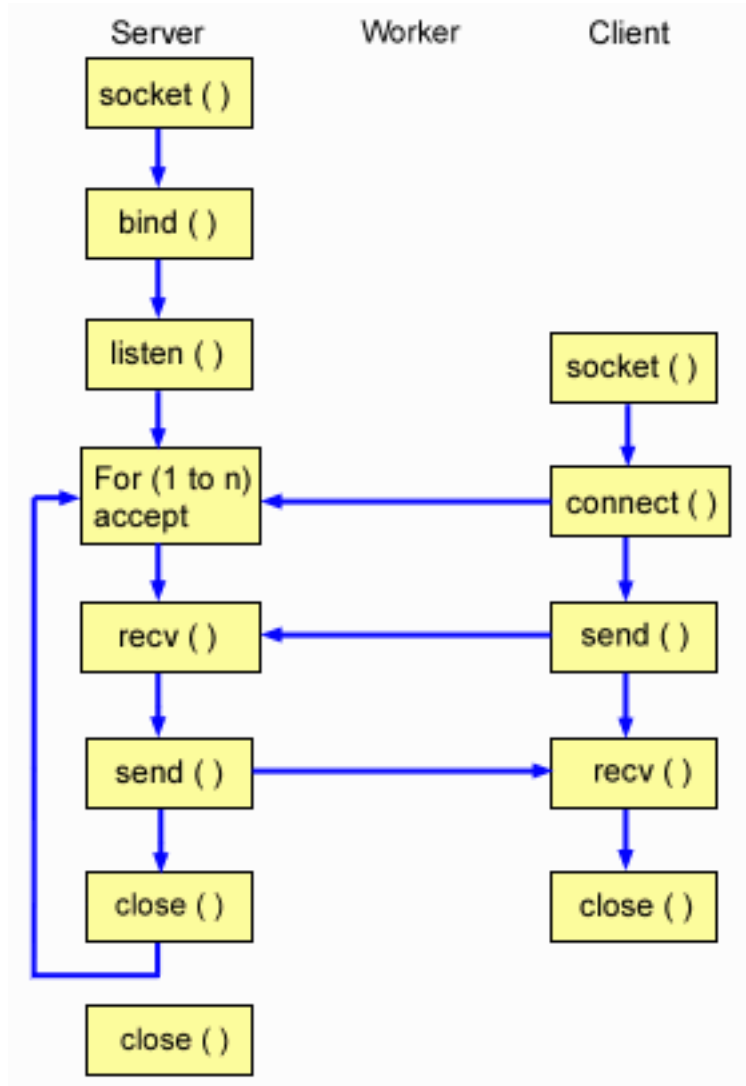
Related information:

`accept()`--Wait for Connection Request and Make Connection API
`spawn()`

Example: Writing an iterative server program

This example illustrates how to create a single server job that handles all incoming connections. When the `accept()` API is completed, the server handles the entire transaction.

The figure illustrates how the server and client jobs interact when the system uses the iterative server design.



Flow of socket events: Iterative server

The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between the server and worker applications. Each set of flows contains links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. The following sequence shows the API calls for the iterative server application:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. After the socket descriptor is created, the `bind()` API gets a unique name for the socket.
3. The `listen()` allows the server to accept incoming client connections.
4. The server uses the `accept()` API to accept an incoming connection request. The `accept()` call blocks indefinitely, waiting for the incoming connection to arrive.
5. The `recv()` API receives data from the client application.
6. The `send()` API echoes the data back to the client.
7. The `close()` API closes any open socket descriptors.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Application creates an iterative server design */
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
    int    i, len, num, rc, on = 1;
    int    listen_sd, accept_sd;
    char    buffer[80];
    struct sockaddr_in6  addr;

    /*****/
    /* If an argument was specified, use it to */
    /* control the number of incoming connections */
    /*****/
    if (argc >= 2)
        num = atoi(argv[1]);
    else
        num = 1;

    /*****/
    /* Create an AF_INET6 stream socket to receive */
    /* incoming connections on */
    /*****/
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }

    /*****/
    /* Allow socket descriptor to be reuseable */
    /*****/
    rc = setsockopt(listen_sd,
                    SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));

    if (rc < 0)
    {
        perror("setsockopt() failed");
        close(listen_sd);
        exit(-1);
    }

    /*****/
    /* Bind the socket */
    /*****/
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family = AF_INET6;
    memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
    addr.sin6_port = htons(SERVER_PORT);
    rc = bind(listen_sd,
              (struct sockaddr *)&addr, sizeof(addr));
    if (rc < 0)
    {
        perror("bind() failed");
        close(listen_sd);
        exit(-1);
    }
}
```

```

}

/*****
/* Set the listen back log */
/*****
rc = listen(listen_sd, 5);
if (rc < 0)
{
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
}

/*****
/* Inform the user that the server is ready */
/*****
printf("The server is ready\n");

/*****
/* Go through the loop once for each connection */
/*****
for (i=0; i < num; i++)
{
    /*****
    /* Wait for an incoming connection */
    /*****
    printf("Iteration: %d\n", i+1);
    printf(" waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
    {
        perror("accept() failed");
        close(listen_sd);
        exit(-1);
    }
    printf(" accept completed successfully\n");

    /*****
    /* Receive a message from the client */
    /*****
    printf(" wait for client to send us a message\n");
    rc = recv(accept_sd, buffer, sizeof(buffer), 0);
    if (rc <= 0)
    {
        perror("recv() failed");
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }
    printf(" <%s>\n", buffer);

    /*****
    /* Echo the data back to the client */
    /*****
    printf(" echo it back\n");
    len = rc;
    rc = send(accept_sd, buffer, len, 0);
    if (rc <= 0)
    {
        perror("send() failed");
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }

    /*****
    /* Close down the incoming connection */
    /*****

```

```

        /*****
        close(accept_sd);
    }

    /*****
    /* Close down the listen socket      */
    /*****
    close(listen_sd);
}

```

Related reference:

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()` operation.

Related information:

`recv()`--Receive Data API

`bind()`--Set Local Address for Socket API

`socket()`--Create Socket API

`listen()`--Invite Incoming Connections Requests API

`accept()`--Wait for Connection Request and Make Connection API

`send()`--Send Data API

`close()`--Close File or Socket Descriptor API

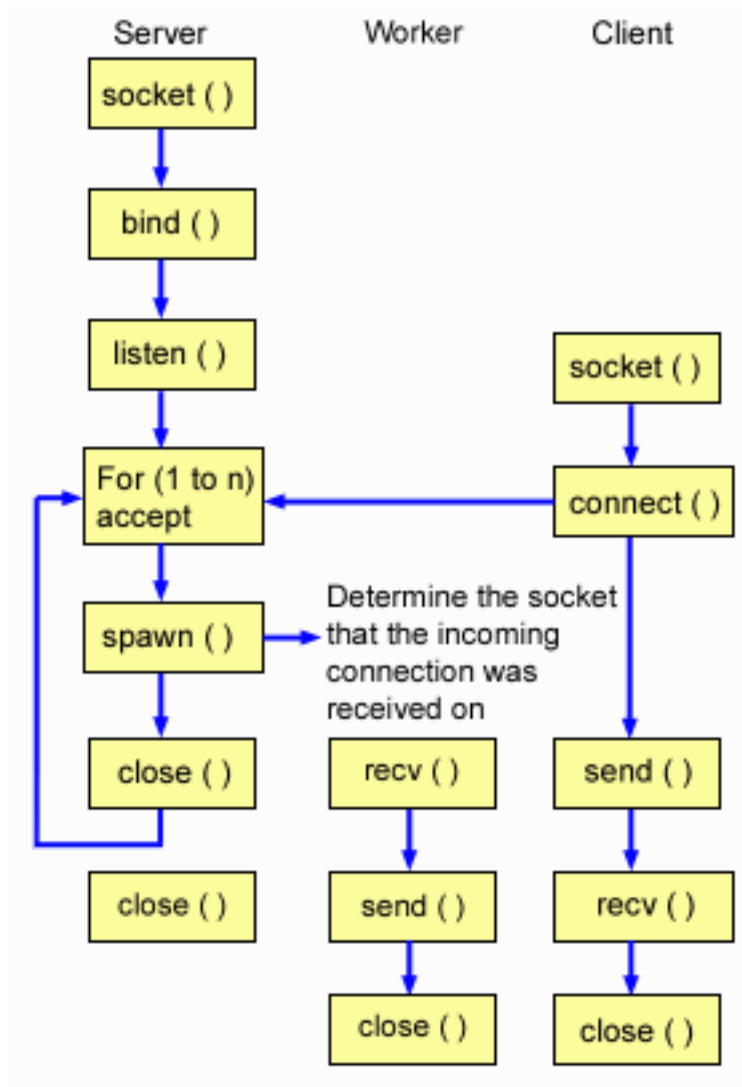
Example: Using the `spawn()` API to create child processes

This example shows how a server program can use the `spawn()` API to create a child process that inherits the socket descriptor from the parent.

The server job waits for an incoming connection, and then calls the `spawn()` API to create children jobs to handle the incoming connection. The child process inherits the following attributes with the `spawn()` API:

- The socket and file descriptors.
- The signal mask.
- The signal action vector.
- The environment variables.

The following figure illustrates how the server, worker, and client jobs interact when the `spawn()` server design is used.



Flow of socket events: Server that uses spawn() to accept and process requests

The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between the server and worker examples. Each set of flows contains links to usage notes on specific APIs. If you need more details about the use of a particular API, you can use these links. The first example uses the following socket calls to create a child process with the spawn() API call:

1. The socket() API returns a socket descriptor, which represents an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (SOCK_STREAM) is used for this socket.
2. After the socket descriptor is created, the bind() API gets a unique name for the socket.
3. The listen() allows the server to accept incoming client connections.
4. The server uses the accept() API to accept an incoming connection request. The accept() call blocks indefinitely, waiting for the incoming connection to arrive.
5. The spawn() API initializes the parameters for a work job to handle incoming requests. In this example, the socket descriptor for the new connection is mapped over to descriptor 0 in the child program.
6. In this example, the first close() API closes the listening socket descriptor. The second close() call ends the accepted socket.

Socket flow of events: Worker job created by spawn()

The second example uses the following sequence of API calls:

1. After the spawn() API is called on the server, the recv() API receives the data from the incoming connection.
2. The send() API echoes data back to the client.
3. The close() API ends the spawned worker job.

Related reference:

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a socket(), connect(), send(), recv(), and close() operation.

Related information:

spawn()

bind()--Set Local Address for Socket API

socket()--Create Socket API

listen()--Invite Incoming Connections Requests API

accept()--Wait for Connection Request and Make Connection API

close()--Close File or Socket Descriptor API

send()--Send Data API

recv()--Receive Data API

Example: Creating a server that uses spawn():

This example shows how to use the spawn() API to create a child process that inherits the socket descriptor from the parent.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/******  
/* Application creates an child process using spawn().          */  
/******  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <spawn.h>  
  
#define SERVER_PORT 12345  
  
main (int argc, char *argv[])  
{  
    int    i, num, pid, rc, on = 1;  
    int    listen_sd, accept_sd;  
    int    spawn_fdmap[1];  
    char   *spawn_argv[1];  
    char   *spawn_envp[1];  
    struct inheritance inherit;  
    struct sockaddr_in6  addr;  
  
    /******  
    /* If an argument was specified, use it to          */  
    /* control the number of incoming connections      */  
    /******  
    if (argc >= 2)  
        num = atoi(argv[1]);  
    else
```



```

    num = 1;

    /******
    /* Create an AF_INET6 stream socket to receive */
    /* incoming connections on */
    /******
listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
if (listen_sd < 0)
{
    perror("socket() failed");
    exit(-1);
}

    /******
    /* Allow socket descriptor to be reuseable */
    /******
rc = setsockopt(listen_sd,
                SOL_SOCKET, SO_REUSEADDR,
                (char *)&on, sizeof(on));

if (rc < 0)
{
    perror("setsockopt() failed");
    close(listen_sd);
    exit(-1);
}

    /******
    /* Bind the socket */
    /******
memset(&addr, 0, sizeof(addr));
addr.sin6_family = AF_INET6;
addr.sin6_port = htons(SERVER_PORT);
memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
    perror("bind() failed");
    close(listen_sd);
    exit(-1);
}

    /******
    /* Set the listen back log */
    /******
rc = listen(listen_sd, 5);
if (rc < 0)
{
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
}

    /******
    /* Inform the user that the server is ready */
    /******
printf("The server is ready\n");

    /******
    /* Go through the loop once for each connection */
    /******
for (i=0; i < num; i++)
{
    /******
    /* Wait for an incoming connection */
    /******
    printf("Iteration: %d\n", i+1);

```

```

printf(" waiting on accept()\n");
accept_sd = accept(listen_sd, NULL, NULL);
if (accept_sd < 0)
{
    perror("accept() failed");
    close(listen_sd);
    exit(-1);
}
printf(" accept completed successfully\n");

/*****
/* Initialize the spawn parameters */
/*
/* The socket descriptor for the new */
/* connection is mapped over to descriptor 0 */
/* in the child program. */
*****/
memset(&inherit, 0, sizeof(inherit));
spawn_argv[0] = NULL;
spawn_envp[0] = NULL;
spawn_fdmmap[0] = accept_sd;

/*****
/* Create the worker job */
*****/
printf(" creating worker job\n");
pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR1.PGM",
            1, spawn_fdmmap, &inherit,
            spawn_argv, spawn_envp);
if (pid < 0)
{
    perror("spawn() failed");
    close(listen_sd);
    close(accept_sd);
    exit(-1);
}
printf(" spawn completed successfully\n");

/*****
/* Close down the incoming connection since */
/* it has been given to a worker to handle */
*****/
close(accept_sd);
}

/*****
/* Close down the listen socket */
*****/
close(listen_sd);
}

```

Related reference:

“Example: Enabling the worker job to receive a data buffer”

This example contains the code that enables the worker job to receive a data buffer from the client job and echo it back.

Example: Enabling the worker job to receive a data buffer:

This example contains the code that enables the worker job to receive a data buffer from the client job and echo it back.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Worker job that receives and echoes back a data buffer to a client */
*****/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
    int    rc, len;
    int    sockfd;
    char    buffer[80];

    /*****/
    /* The descriptor for the incoming connection is */
    /* passed to this worker job as a descriptor 0. */
    /*****/
    sockfd = 0;

    /*****/
    /* Receive a message from the client */
    /*****/
    printf("Wait for client to send us a message\n");
    rc = recv(sockfd, buffer, sizeof(buffer), 0);
    if (rc <= 0)
    {
        perror("recv() failed");
        close(sockfd);
        exit(-1);
    }
    printf("<%s>\n", buffer);

    /*****/
    /* Echo the data back to the client */
    /*****/
    printf("Echo it back\n");
    len = rc;
    rc = send(sockfd, buffer, len, 0);
    if (rc <= 0)
    {
        perror("send() failed");
        close(sockfd);
        exit(-1);
    }

    /*****/
    /* Close down the incoming connection */
    /*****/
    close(sockfd);
}

```

Related reference:

“Example: Creating a server that uses spawn()” on page 98

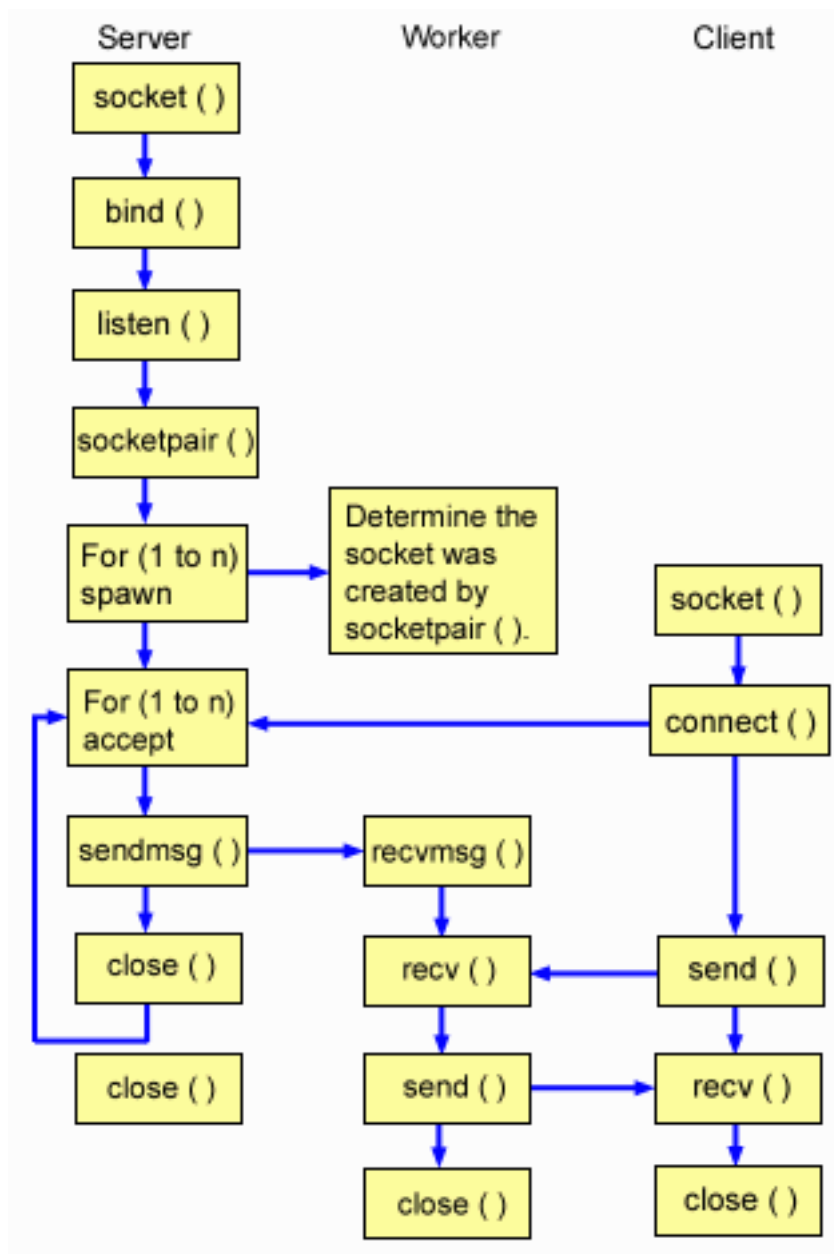
This example shows how to use the spawn() API to create a child process that inherits the socket descriptor from the parent.

Example: Passing descriptors between processes

These examples demonstrate how to design a server program using the sendmsg() and recvmsg() APIs to handle incoming connections.

When the server starts, it creates a pool of worker jobs. These preallocated (spawned) worker jobs wait until needed. When the client job connects to the server, the server gives the incoming connection to one of the worker jobs.

The following figure illustrates how the server, worker, and client jobs interact when the system uses the sendmsg() and recvmsg() server design.



Flow of socket events: Server that uses sendmsg() and recvmsg() APIs

The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between the server and worker examples. The first example uses the following socket calls to create a child process with the sendmsg() and recvmsg() API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (SOCK_STREAM) is used for this socket.
2. After the socket descriptor is created, the `bind()` API gets a unique name for the socket.
3. The `listen()` allows the server to accept incoming client connections.
4. The `socketpair()` API creates a pair of UNIX datagram sockets. A server can use the `socketpair()` API to create a pair of AF_UNIX sockets.

5. The `spawn()` API initializes the parameters for a work job to handle incoming requests. In this example, the child job created inherits the socket descriptor that was created by the `socketpair()`.
6. The server uses the `accept()` API to accept an incoming connection request. The `accept()` call blocks indefinitely, waiting for the incoming connection to arrive.
7. The `sendmsg()` API sends an incoming connection to one of the worker jobs. The child process accepts the connection with `recvmsg()` API. The child job is not active when the server called `sendmsg()`.
8. In this example, the first `close()` API closes the accepted socket. The second `close()` call ends the listening socket.

Socket flow of events: Worker job that uses `recvmsg()`

The second example uses the following sequence of API calls:

1. After the server has accepted a connection and passed its socket descriptor to the worker job, the `recvmsg()` API receives the descriptor. In this example, the `recvmsg()` API waits until the server sends the descriptor.
2. The `recv()` API receives a message from the client.
3. The `send()` API echoes data back to the client.
4. The `close()` API ends the worker job.

Related reference:

“Descriptor passing between processes: `sendmsg()` and `recvmsg()`” on page 74

Passing an open descriptor between jobs allows one process (typically a server) to do everything that is required to obtain the descriptor, such as opening a file, establishing a connection, and waiting for the `accept()` API to complete. It also allows another process (typically a worker) to handle all the data transfer operations as soon as the descriptor is open.

“Socket application design recommendations” on page 88

Before working with a socket application, assess the functional requirements, goals, and needs of the socket application. Also, consider the performance requirements and the system resource impacts of the application.

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()` operation.

Related information:

`spawn()`

`bind()`--Set Local Address for Socket API

`socket()`--Create Socket API

`listen()`--Invite Incoming Connections Requests API

`accept()`--Wait for Connection Request and Make Connection API

`close()`--Close File or Socket Descriptor API

`socketpair()`--Create a Pair of Sockets API

--Send a Message Over a Socket API

`recvmsg()`--Receive a Message Over a Socket API

`send()`--Send Data API

`recv()`--Receive Data API

Example: Server program used for `sendmsg()` and `recvmsg()`:

This example shows how to use the `sendmsg()` API to create a pool of worker jobs.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Server example that uses sendmsg() to create worker jobs      */
/*****
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
    int    i, num, pid, rc, on = 1;
    int    listen_sd, accept_sd;
    int    server_sd, worker_sd, pair_sd[2];
    int    spawn_fdmap[1];
    char    *spawn_argv[1];
    char    *spawn_envp[1];
    struct inheritance    inherit;
    struct msghdr    msg;
    struct sockaddr_in6    addr;

    /*****/
    /* If an argument was specified, use it to      */
    /* control the number of incoming connections */
    /*****/
    if (argc >= 2)
        num = atoi(argv[1]);
    else
        num = 1;

    /*****/
    /* Create an AF_INET6 stream socket to receive */
    /* incoming connections on                     */
    /*****/
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }

    /*****/
    /* Allow socket descriptor to be reuseable     */
    /*****/
    rc = setsockopt(listen_sd,
                    SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));

    if (rc < 0)
    {
        perror("setsockopt() failed");
        close(listen_sd);
        exit(-1);
    }

    /*****/
    /* Bind the socket                             */
    /*****/
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family    = AF_INET6;
    memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
    addr.sin6_port      = htons(SERVER_PORT);
    rc = bind(listen_sd,
              (struct sockaddr *)&addr, sizeof(addr));
    if (rc < 0)
    {

```

```

        perror("bind() failed");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Set the listen back log */
    /******
    rc = listen(listen_sd, 5);
    if (rc < 0)
    {
        perror("listen() failed");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Create a pair of UNIX datagram sockets */
    /******
    rc = socketpair(AF_UNIX, SOCK_DGRAM, 0, pair_sd);
    if (rc != 0)
    {
        perror("socketpair() failed");
        close(listen_sd);
        exit(-1);
    }
    server_sd = pair_sd[0];
    worker_sd = pair_sd[1];

    /******
    /* Initialize parms before entering for loop */
    /*
    /* The worker socket descriptor is mapped to */
    /* descriptor 0 in the child program. */
    /******
    memset(&inherit, 0, sizeof(inherit));
    spawn_argv[0] = NULL;
    spawn_envp[0] = NULL;
    spawn_fdmapp[0] = worker_sd;

    /******
    /* Create each of the worker jobs */
    /******
    printf("Creating worker jobs...\n");
    for (i=0; i < num; i++)
    {
        pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR2.PGM",
                    1, spawn_fdmapp, &inherit,
                    spawn_argv, spawn_envp);
        if (pid < 0)
        {
            perror("spawn() failed");
            close(listen_sd);
            close(server_sd);
            close(worker_sd);
            exit(-1);
        }
        printf(" Worker = %d\n", pid);
    }

    /******
    /* Close down the worker side of the socketpair */
    /******
    close(worker_sd);

    /******
    /* Inform the user that the server is ready */

```

```

/*****/
printf("The server is ready\n");

/*****/
/* Go through the loop once for each connection */
/*****/
for (i=0; i < num; i++)
{
    /*****/
    /* Wait for an incoming connection */
    /*****/
    printf("Iteration: %d\n", i+1);
    printf(" waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
    {
        perror("accept() failed");
        close(listen_sd);
        close(server_sd);
        exit(-1);
    }
    printf(" accept completed successfully\n");

    /*****/
    /* Initialize message header structure */
    /*****/
    memset(&msg, 0, sizeof(msg));

    /*****/
    /* We are not sending any data so we do not */
    /* need to set either of the msg_iov fields. */
    /* The memset of the message header structure */
    /* will set the msg_iov pointer to NULL and */
    /* it will set the msg_iovcnt field to 0. */
    /*****/

    /*****/
    /* The only fields in the message header */
    /* structure that need to be filled in are */
    /* the msg_accrighs fields. */
    /*****/
    msg.msg_accrighs = (char *)&accept_sd;
    msg.msg_accrighslen = sizeof(accept_sd);

    /*****/
    /* Give the incoming connection to one of the */
    /* worker jobs. */
    /* */
    /* NOTE: We do not know which worker job will */
    /* get this inbound connection. */
    /*****/
    rc = sendmsg(server_sd, &msg, 0);
    if (rc < 0)
    {
        perror("sendmsg() failed");
        close(listen_sd);
        close(accept_sd);
        close(server_sd);
        exit(-1);
    }
    printf(" sendmsg completed successfully\n");

    /*****/
    /* Close down the incoming connection since */
    /* it has been given to a worker to handle */
    /*****/
    close(accept_sd);

```



```

    }

    /*****
    /* Close down the server and listen sockets      */
    /*****/
    close(server_sd);
    close(listen_sd);
}

```

Related reference:

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a socket(), connect(), send(), recv(), and close() operation.

Example: Worker program used for sendmsg() and recvmsg():

This example shows how to use the recvmsg() API client job to receive the worker jobs.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****/
/* Worker job that uses the recvmsg to process client requests      */
/*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
    int    rc, len;
    int    worker_sd, pass_sd;
    char    buffer[80];
    struct iovec    iov[1];
    struct msghdr    msg;

    /*****/
    /* One of the socket descriptors that was      */
    /* returned by socketpair(), is passed to this      */
    /* worker job as descriptor 0.      */
    /*****/
    worker_sd = 0;

    /*****/
    /* Initialize message header structure      */
    /*****/
    memset(&msg,    0, sizeof(msg));
    memset(iov,    0, sizeof(iov));

    /*****/
    /* The recvmsg() call will NOT block unless a      */
    /* non-zero length data buffer is specified      */
    /*****/
    iov[0].iov_base = buffer;
    iov[0].iov_len = sizeof(buffer);
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    /*****/
    /* Fill in the msg_accrighs fields so that we      */
    /* can receive the descriptor      */
    /*****/
    msg.msg_accrighs = (char *)&pass_sd;
    msg.msg_accrighslen = sizeof(pass_sd);

    /*****/

```

```

/* Wait for the descriptor to arrive */
/*****
printf("Waiting on recvmmsg\n");
rc = recvmmsg(worker_sd, &msg, 0);
if (rc < 0)
{
    perror("recvmmsg() failed");
    close(worker_sd);
    exit(-1);
}
else if (msg.msg_accrightrightlen <= 0)
{
    printf("Descriptor was not received\n");
    close(worker_sd);
    exit(-1);
}
else
{
    printf("Received descriptor = %d\n", pass_sd);
}

/*****
/* Receive a message from the client */
/*****
printf("Wait for client to send us a message\n");
rc = recv(pass_sd, buffer, sizeof(buffer), 0);
if (rc <= 0)
{
    perror("recv() failed");
    close(worker_sd);
    close(pass_sd);
    exit(-1);
}
printf("<%s>\n", buffer);

/*****
/* Echo the data back to the client */
/*****
printf("Echo it back\n");
len = rc;
rc = send(pass_sd, buffer, len, 0);
if (rc <= 0)
{
    perror("send() failed");
    close(worker_sd);
    close(pass_sd);
    exit(-1);
}

/*****
/* Close down the descriptors */
/*****
close(worker_sd);
close(pass_sd);
}

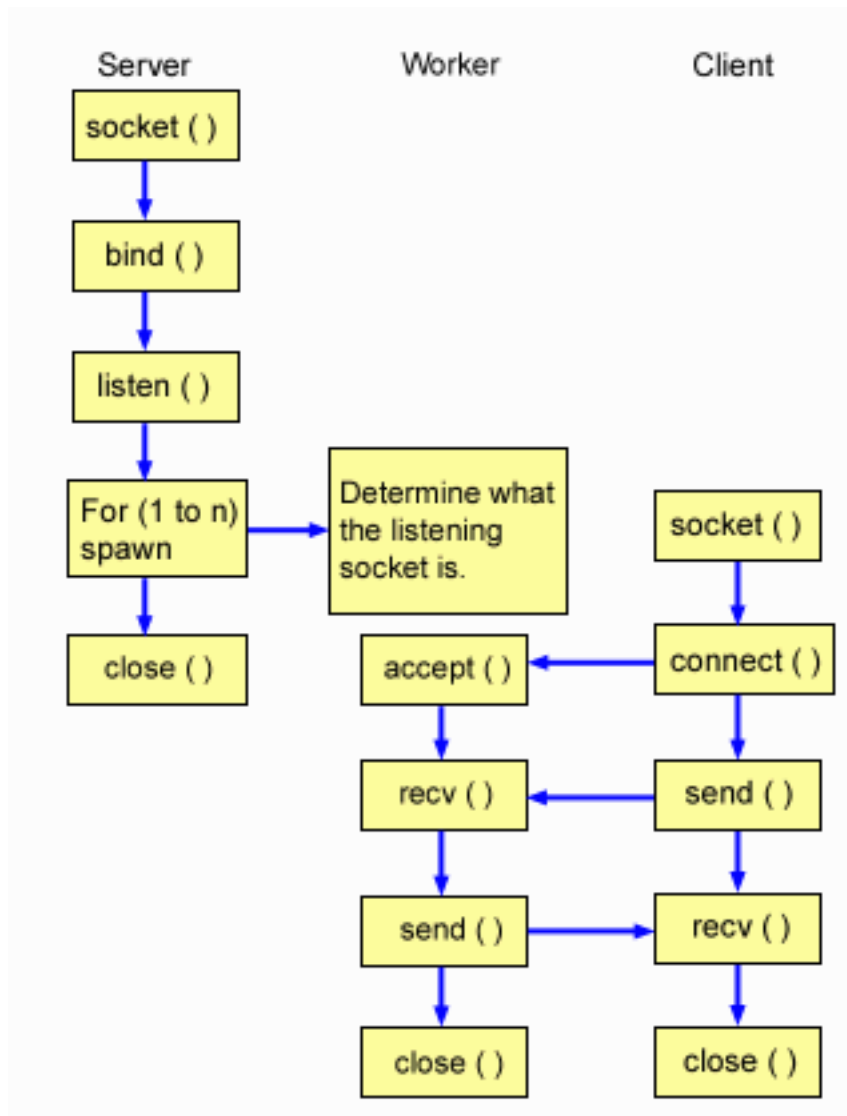
```

Examples: Using multiple accept() APIs to handle incoming requests

These examples show how to design a server program that uses the multiple accept() model for handling incoming connection requests.

When the multiple accept() server starts up, it does a socket(), bind(), and listen() as normal. It then creates a pool of worker jobs and gives each worker job the listening socket. Each multiple accept() worker then calls accept().

The following figure illustrates how the server, worker, and client jobs interact when the system uses the multiple accept() server design.



Flow of socket events: Server that creates a pool of multiple accept() worker jobs

The following sequence of the socket calls provides a description of the figure. It also describes the relationship between the server and worker examples. Each set of flows contains links to usage notes on specific APIs. If you need more details about the use of a particular API, you can use these links. The first example uses the following socket calls to create a child process:

1. The socket() API returns a socket descriptor, which represents an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (SOCK_STREAM) is used for this socket.
2. After the socket descriptor is created, the bind() API gets a unique name for the socket.
3. The listen() API allows the server to accept incoming client connections.
4. The spawn() API creates each of the worker jobs.
5. In this example, the first close() API closes the listening socket.

Socket flow of events: Worker job that multiple accept()

The second example uses the following sequence of API calls:

1. After the server has spawned the worker jobs, the listen socket descriptor is passed to this worker job as a command line parameter. The accept() API waits for an incoming client connection.
2. The recv() API receives a message from the client.
3. The send() API echoes data back to the client.
4. The close() API ends the worker job.

Related reference:

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a socket(), connect(), send(), recv(), and close() operation.

Related information:

spawn()

bind()--Set Local Address for Socket API

socket()--Create Socket API

listen()--Invite Incoming Connections Requests API

close()--Close File or Socket Descriptor API

accept()--Wait for Connection Request and Make Connection API

send()--Send Data API

recv()--Receive Data API

Example: Server program to create a pool of multiple accept() worker jobs:

This example shows how to use the multiple accept() model to create a pool of worker jobs.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/******  
/* Server example creates a pool of worker jobs with multiple accept() calls */  
/******  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <spawn.h>  
  
#define SERVER_PORT 12345  
  
main (int argc, char *argv[])  
{  
    int    i, num, pid, rc, on = 1;  
    int    listen_sd, accept_sd;  
    int    spawn_fdmap[1];  
    char   *spawn_argv[1];  
    char   *spawn_envp[1];  
    struct inheritance inherit;  
    struct sockaddr_in6 addr;  
  
    /******  
    /* If an argument was specified, use it to      */  
    /* control the number of incoming connections  */  
    /******  
    if (argc >= 2)  
        num = atoi(argv[1]);  
    else
```

```

    num = 1;

    /******
    /* Create an AF_INET6 stream socket to receive */
    /* incoming connections on */
    /******
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }

    /******
    /* Allow socket descriptor to be reuseable */
    /******
    rc = setsockopt(listen_sd,
                    SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));

    if (rc < 0)
    {
        perror("setsockopt() failed");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Bind the socket */
    /******
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family = AF_INET6;
    memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
    addr.sin6_port = htons(SERVER_PORT);
    rc = bind(listen_sd,
              (struct sockaddr *)&addr, sizeof(addr));
    if (rc < 0)
    {
        perror("bind() failed");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Set the listen back log */
    /******
    rc = listen(listen_sd, 5);
    if (rc < 0)
    {
        perror("listen() failed");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Initialize parameters before entering for loop */
    /* */
    /* The listen socket descriptor is mapped to */
    /* descriptor 0 in the child program. */
    /******
    memset(&inherit, 0, sizeof(inherit));
    spawn_argv[0] = NULL;
    spawn_envp[0] = NULL;
    spawn_fdmmap[0] = listen_sd;

    /******
    /* Create each of the worker jobs */
    /******

```

```

printf("Creating worker jobs...\n");
for (i=0; i < num; i++)
{
    pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR3.PGM",
                1, spawn_fdmapi, &inherit,
                spawn_argv, spawn_envp);
    if (pid < 0)
    {
        perror("spawn() failed");
        close(listen_sd);
        exit(-1);
    }
    printf(" Worker = %d\n", pid);
}

/*****
/* Inform the user that the server is ready */
*****/
printf("The server is ready\n");

/*****
/* Close down the listening socket */
*****/
close(listen_sd);
}

```

Related reference:

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a socket(), connect(), send(), recv(), and close() operation.

Example: Worker jobs for multiple accept():

This example shows how multiple accept() APIs receive the worker jobs and call the accept() server.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Worker job uses multiple accept() to handle incoming client connections*/
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
    int    rc, len;
    int    listen_sd, accept_sd;
    char    buffer[80];

    /*****
    /* The listen socket descriptor is passed to
    /* this worker job as a command line parameter
    *****/
    listen_sd = 0;

    /*****
    /* Wait for an incoming connection
    *****/
    printf("Waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
    {
        perror("accept() failed");
        close(listen_sd);
    }
}

```

```

        exit(-1);
    }
    printf("Accept completed successfully\n");

    /******
    /* Receive a message from the client */
    /******
    printf("Wait for client to send us a message\n");
    rc = recv(accept_sd, buffer, sizeof(buffer), 0);
    if (rc <= 0)
    {
        perror("recv() failed");
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }
    printf("<%=s>\n", buffer);

    /******
    /* Echo the data back to the client */
    /******
    printf("Echo it back\n");
    len = rc;
    rc = send(accept_sd, buffer, len, 0);
    if (rc <= 0)
    {
        perror("send() failed");
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }

    /******
    /* Close down the descriptors */
    /******
    close(listen_sd);
    close(accept_sd);
}

```

Example: Generic client

This example contains the code for a common client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()` operation.

The client job is not aware that the data buffer it sent and received is going to a worker job rather than the server. If you want to create a client application that works whether the server uses the `AF_INET` address family or `AF_INET6` address family, use the IPv4 or IPv6 client example.

This client job works with each of these common connection-oriented server designs:

- An iterative server. See Example: Writing an iterative server program.
- A spawn server and worker. See Example: Using the `spawn()` API to create child processes.
- A `sendmsg()` server and `rcvmsg()` worker. See Example: Server program used for `sendmsg()` and `rcvmsg()`.
- A multiple `accept()` design. See Example: Server program to create a pool of multiple `accept()` worker jobs.
- A nonblocking I/O and `select()` design. See Example: Nonblocking I/O and `select()`.
- A server that accepts connections from either an IPv4 or IPv6 client. See Example: Accepting connections from both IPv6 and IPv4 clients.

Socket flow of events: Generic client

The following example program uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. After the socket descriptor is received, the `connect()` API is used to establish a connection to the server.
3. The `send()` API sends the data buffer to the worker jobs.
4. The `recv()` API receives the data buffer from the worker jobs.
5. The `close()` API closes any open socket descriptors.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Generic client example is used with connection-oriented server designs */
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
    int    len, rc;
    int    sockfd;
    char    send_buf[80];
    char    recv_buf[80];
    struct sockaddr_in6  addr;

    /*****/
    /* Create an AF_INET6 stream socket */
    /*****/
    sockfd = socket(AF_INET6, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("socket");
        exit(-1);
    }

    /*****/
    /* Initialize the socket address structure */
    /*****/
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family = AF_INET6;
    memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
    addr.sin6_port = htons(SERVER_PORT);

    /*****/
    /* Connect to the server */
    /*****/
    rc = connect(sockfd,
                (struct sockaddr *)&addr,
                sizeof(struct sockaddr_in6));
    if (rc < 0)
    {
        perror("connect");
        close(sockfd);
        exit(-1);
    }
    printf("Connect completed.\n");

    /*****/
    /* Enter data buffer that is to be sent */
    */

```



```

/*****/
printf("Enter message to be sent:\n");
gets(send_buf);

/*****/
/* Send data buffer to the worker job */
/*****/
len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
if (len != strlen(send_buf) + 1)
{
    perror("send");
    close(sockfd);
    exit(-1);
}
printf("%d bytes sent\n", len);

/*****/
/* Receive data buffer from the worker job */
/*****/
len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
if (len != strlen(send_buf) + 1)
{
    perror("recv");
    close(sockfd);
    exit(-1);
}
printf("%d bytes received\n", len);

/*****/
/* Close down the socket */
/*****/
close(sockfd);
}

```

Related reference:

“Example: IPv4 or IPv6 client” on page 85

This sample program can be used with the server application that accepts requests from either IPv4 or IPv6 clients.

“Examples: Connection-oriented designs” on page 91

You can design a connection-oriented socket server on the system in a number of ways. These example programs can be used to create your own connection-oriented designs.

“Example: Writing an iterative server program” on page 92

This example illustrates how to create a single server job that handles all incoming connections. When the `accept()` API is completed, the server handles the entire transaction.

“Example: Passing descriptors between processes” on page 101

These examples demonstrate how to design a server program using the `sendmsg()` and `recvmsg()` APIs to handle incoming connections.

“Example: Server program used for `sendmsg()` and `recvmsg()`” on page 103

This example shows how to use the `sendmsg()` API to create a pool of worker jobs.

“Examples: Using multiple `accept()` APIs to handle incoming requests” on page 108

These examples show how to design a server program that uses the multiple `accept()` model for handling incoming connection requests.

“Example: Server program to create a pool of multiple `accept()` worker jobs” on page 110

This example shows how to use the multiple `accept()` model to create a pool of worker jobs.

“Example: Using the `spawn()` API to create child processes” on page 96

This example shows how a server program can use the `spawn()` API to create a child process that inherits the socket descriptor from the parent.

“Example: Accepting connections from both IPv6 and IPv4 clients” on page 80

This example program demonstrates how to create a server/client model that accepts requests from both IPv4 (those socket applications that use the `AF_INET` address family) and IPv6 (those applications that

use the AF_INET6 address family).

“Example: Using asynchronous I/O”

An application creates an I/O completion port using the `QsoCreateIOCompletionPort()` API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests.

“Example: Nonblocking I/O and `select()`” on page 152

This sample program illustrates a server application that uses nonblocking and the `select()` API.

Related information:

`socket()`--Create Socket API

`connect()`--Establish Connection or Destination Address API

`close()`--Close File or Socket Descriptor API

`send()`--Send Data API

`recv()`--Receive Data API

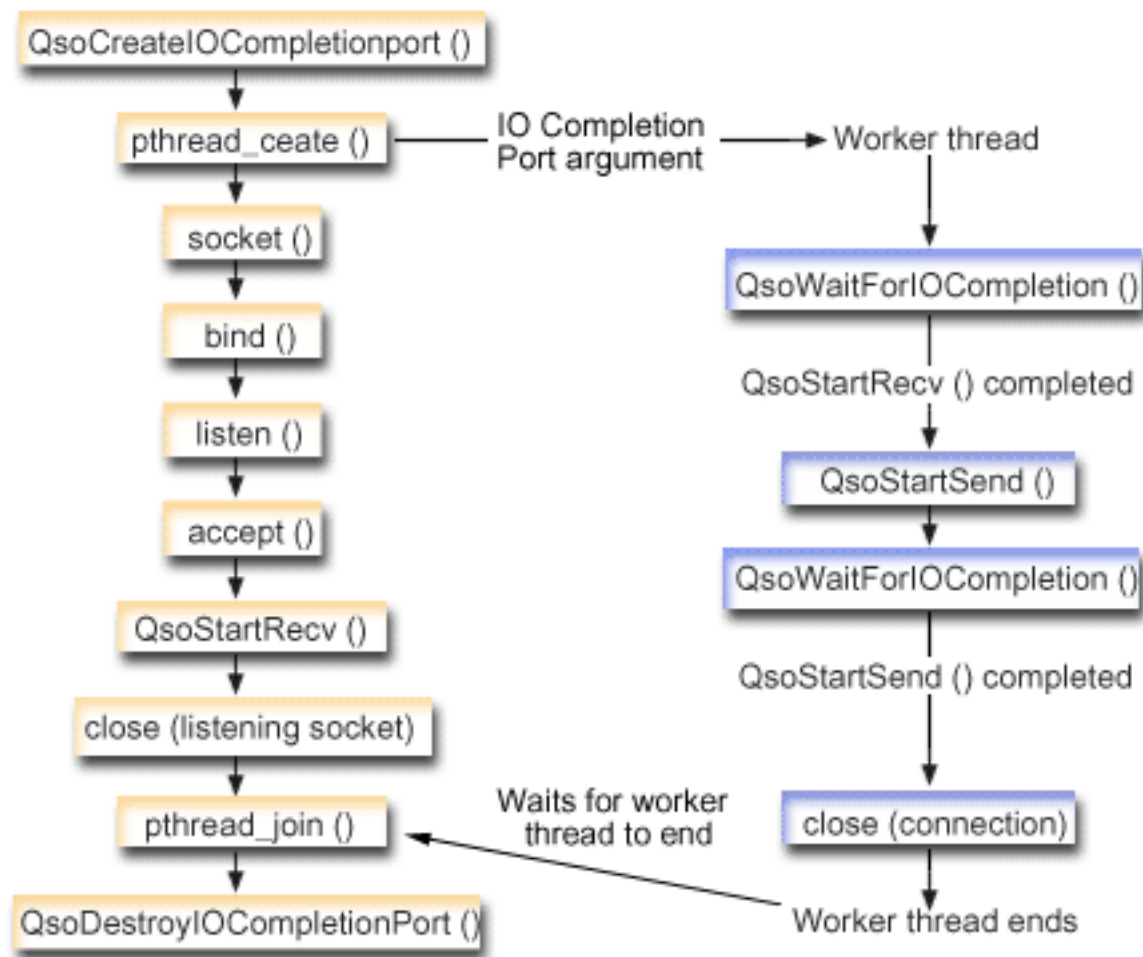
Example: Using asynchronous I/O

An application creates an I/O completion port using the `QsoCreateIOCompletionPort()` API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests.

The application starts an input or an output function, specifying an I/O completion port handle. When the I/O is completed, status information and an application-defined handle are posted to the specified I/O completion port. The post to the I/O completion port wakes up exactly one of possibly many threads that are waiting. The application receives the following items:

- A buffer that was supplied on the original request
- The length of data that was processed to or from that buffer
- A indication of what type of I/O operation has been completed
- Application-defined handle that was passed on the initial I/O request

This application handle can be the socket descriptor identifying the client connection, or a pointer to storage that contains extensive information about the state of the client connection. Since the operation was completed and the application handle was passed, the worker thread determines the next step to complete the client connection. Worker threads that process these completed asynchronous operations can handle many different client requests and are not tied to just one. Because copying to and from user buffers occurs asynchronously to the server processes, wait time for client request diminishes. This can be beneficial on systems where there are multiple processors.



Flow of socket events: Asynchronous I/O server

The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between the server and worker examples. Each set of flows contain links to usage notes on specific APIs. If you need more details on the use of a particular API, you can use these links. This flow describes the socket calls in the following sample application. Use this server example with the generic client example.

1. Master thread creates I/O completion port by calling `QsoCreateIOCompletionPort()`
2. Master thread creates pool of worker thread(s) to process any I/O completion port requests with the `pthread_create` function.
3. Worker thread(s) call `QsoWaitForIOCompletionPort()` which waits for client requests to process.
4. The master thread accepts a client connection and proceeds to issue a `QsoStartRecv()` which specifies the I/O completion port upon which the worker threads are waiting.

Note: You can also use `accept` asynchronously by using the `QsoStartAccept()`.

5. At some point, a client request arrives asynchronous to the server process. The sockets operating system loads the supplied user buffer and sends the completed `QsoStartRecv()` request to the specified I/O completion port. One worker thread is awoken and proceeds to process this request.
6. The worker thread extracts the client socket descriptor from the application-defined handle and proceeds to echo the received data back to the client by performing a `QsoStartSend()` operation.

7. If the data can be immediately sent, then the QsoStartSend() API returns indication of the fact; otherwise, the sockets operating system sends the data as soon as possible and posts indication of the fact to the specified I/O completion port. The worker thread gets indication of data sent and can wait on the I/O completion port for another request or end if instructed to do so. The QsoPostIOCompletion() API can be used by the master thread to post a worker thread end event.
8. Master thread waits for worker thread to finish and then destroys the I/O completion port by calling the QsoDestroyIOCompletionPort() API.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <unistd.h>
#define _MULTI_THREADED
#include "pthread.h"
#include "qsoasync.h"
#define BufferLength 80
#define Failure 0
#define Success 1
#define SERVPOR 12345

void *workerThread(void *arg);

/*****
/*
/* Function Name: main
/*
/* Descriptive Name: Master thread will establish a client
/* connection and hand processing responsibility
/* to a worker thread.
/* Note: Due to the thread attribute of this program, spawn() must
/* be used to invoke.
*****/

int main()
{
    int listen_sd, client_sd, rc;
    int on = 1, ioCompPort;
    pthread_t thr;
    void *status;
    char buffer[BufferLength];
    struct sockaddr_in6 serveraddr;
    Qso_OverlappedIO_t ioStruct;

    /*****/
    /* Create an I/O completion port for this
    /* process.
    *****/
    if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
    {
        perror("QsoCreateIOCompletionPort() failed");
        exit(-1);
    }

    /*****/
    /* Create a worker thread
    /* to process all client requests. The
    /* worker thread will wait for client
    *****/
```

```

/* requests to arrive on the I/O completion */
/* port just created. */
/*****
rc = pthread_create(&thr, NULL, workerThread,
                    &ioCompPort);

if (rc < 0)
{
    perror("pthread_create() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    exit(-1);
}

/*****
/* Create an AF_INET6 stream socket to */
/* receive incoming connections on */
/*****
if ((listen_sd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
{
    perror("socket() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    exit(-1);
}

/*****
/* Allow socket descriptor to be reuseable */
/*****
if ((rc = setsockopt(listen_sd, SOL_SOCKET,
                    SO_REUSEADDR,
                    (char *)&on,
                    sizeof(on))) < 0)
{
    perror("setsockopt() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    exit(-1);
}

/*****
/* bind the socket */
/*****
memset(&serveraddr, 0x00, sizeof(struct sockaddr_in6));
serveraddr.sin6_family = AF_INET6;
serveraddr.sin6_port = htons(SERVPORT);
memcpy(&serveraddr.sin6_addr, &in6addr_any, sizeof(in6addr_any));

if ((rc = bind(listen_sd,
                (struct sockaddr *)&serveraddr,
                sizeof(serveraddr))) < 0)
{
    perror("bind() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    exit(-1);
}

/*****
/* Set listen backlog */
/*****
if ((rc = listen(listen_sd, 10)) < 0)
{
    perror("listen() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    exit(-1);
}

```

```

printf("Waiting for client connection.\n");

/*****/
/* accept an incoming client connection. */
/*****/
if ((client_sd = accept(listen_sd, (struct sockaddr *)NULL,
                        NULL)) < 0)
{
    perror("accept() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    exit(-1);
}

/*****/
/* Issue QsoStartRecv() to receive client */
/* request. */
/* Note: */
/* postFlag == on denoting request should */
/* posted to the I/O */
/* completion port, even if */
/* if request is immediately */
/* available. Worker thread */
/* will process client */
/* request. */
/*****/

/*****/
/* initialize Qso_OverlappedIO_t structure - */
/* reserved fields must be hex 00's. */
/*****/
memset(&ioStruct, '\0', sizeof(ioStruct));

ioStruct.buffer = buffer;
ioStruct.bufferLength = sizeof(buffer);

/*****/
/* Store the client descriptor in the */
/* Qso_OverlappedIO_t descriptorHandle field.*/
/* This area is used to house information */
/* defining the state of the client */
/* connection. Field descriptorHandle is */
/* defined as a (void *) to allow the server */
/* to address more extensive client */
/* connection state if needed. */
/*****/
*((int*)&ioStruct.descriptorHandle) = client_sd;
ioStruct.postFlag = 1;
ioStruct.fillBuffer = 0;

rc = QsoStartRecv(client_sd, ioCompPort, &ioStruct);
if (rc == -1)
{
    perror("QsoStartRecv() failed");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    close(client_sd);
    exit(-1);
}
/*****/
/* close the server's listening socket. */
/*****/
close(listen_sd);

/*****/
/* Wait for worker thread to finish */
/*****/

```

```

        /* processing client connection.          */
        /*******/
        rc = pthread_join(thr, &status);

        QsoDestroyIOCompletionPort(ioCompPort);
        if ( rc == 0 && (rc = __INT(status)) == Success)
        {
            printf("Success.\n");
            exit(0);
        }
        else
        {
            perror("pthread_join() reported failure");
            exit(-1);
        }
    }
    /* end workerThread */

    /*******/
    /*
    /* Function Name: workerThread
    /*
    /* Descriptive Name: Process client connection.
    /*
    /*******/
    void *workerThread(void *arg)
    {
        struct timeval waitTime;
        int ioCompPort, clientfd;
        Qso_OverlappedIO_t ioStruct;
        int rc, tID;
        pthread_t thr;
        pthread_id_np_t t_id;
        t_id = pthread_getthreadid_np();
        tID = t_id.intId.lo;

        /*******/
        /* I/O completion port is passed to this */
        /* routine.
        /*
        /*******/
        ioCompPort = *(int *)arg;

        /*******/
        /* Wait on the supplied I/O completion port */
        /* for a client request.
        /*
        /*******/
        waitTime.tv_sec = 500;
        waitTime.tv_usec = 0;
        rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
        if (rc == 1 && ioStruct.returnValue != -1)
        /*******/
        /* Client request has been received.
        /*
        /*******/
        ;
        else
        {
            printf("QsoWaitForIOCompletion() or QsoStartRecv() failed.\n");
            if(rc != 1)
                perror("QsoWaitForIOCompletion() failed");
            if(ioStruct.returnValue == -1)
                printf("QsoStartRecv() failed - %s\n",
                    strerror(ioStruct.errnoValue));

            return __VOID(Failure);
        }
    }

```

```

/*****
/* Obtain the socket descriptor associated */
/* with the client connection. */
/*****
clientfd = *((int *) &ioStruct.descriptorHandle);

/*****
/* Echo the data back to the client. */
/* Note: postFlag == 0. If write completes */
/* immediate then indication will be */
/* returned, otherwise once the */
/* write is performed the I/O Completion */
/* port will be posted. */
/*****
ioStruct.postFlag = 0;
ioStruct.bufferLength = ioStruct.returnValue;
rc = QsoStartSend(clientfd, ioCompPort, &ioStruct);

if (rc == 0)
/*****
/* Operation complete - data has been sent. */
/*****
;
else
{
/*****
/* Two possibilities */
/* rc == -1 */
/* Error on function call */
/* rc == 1 */
/* Write cannot be immediately */
/* performed. Once complete, the I/O */
/* completion port will be posted. */
/*****

if (rc == -1)
{
printf("QsoStartSend() failed.\n");
perror("QsoStartSend() failed");
close(clientfd);
return __VOID(Failure);
}
/*****
/* Wait for operation to complete. */
/*****
rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
if (rc == 1 && ioStruct.returnValue != -1)
/*****
/* Send successful. */
/*****
;
else
{
printf("QsoWaitForIOCompletion() or QsoStartSend() failed.\n");
if(rc != 1)
perror("QsoWaitForIOCompletion() failed");
if(ioStruct.returnValue == -1)
printf("QsoStartRecv() failed - %s\n",
strerror(ioStruct.errnoValue));
return __VOID(Failure);
}
}
close(clientfd);
return __VOID(Success);
} /* end workerThread */

```

Related concepts:

“Asynchronous I/O” on page 43

Asynchronous I/O APIs provide a method for threaded client/server models to perform highly concurrent and memory-efficient I/O.

Related reference:

“Socket application design recommendations” on page 88

Before working with a socket application, assess the functional requirements, goals, and needs of the socket application. Also, consider the performance requirements and the system resource impacts of the application.

“Examples: Connection-oriented designs” on page 91

You can design a connection-oriented socket server on the system in a number of ways. These example programs can be used to create your own connection-oriented designs.

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()` operation.

“Example: Using signals with blocking socket APIs” on page 164

When a process or an application becomes blocked, signals allow you to be notified. They also provide a time limit for blocking processes.

Related information:

`QsoCreateIOCompletionPort()`--Create I/O Completion Port API

`pthread_create`

`QsoWaitForIOCompletion()`--Wait for I/O Operation API

`QsoStartAccept()`--Start asynchronous accept operation API

`QsoStartSend()`--Start Asynchronous Send Operation API

`QsoDestroyIOCompletionPort()`--Destroy I/O Completion Port API

Examples: Establishing secure connections

You can create secure server and clients using either the Global Security Kit (GSKit) APIs or the Secure Sockets Layer (SSL_) APIs.

GSKit APIs are preferred because they are supported across IBM systems, while SSL_ APIs exist only in the IBM i operating system. Each set of Secure Sockets APIs has return codes that help you identify errors when establishing secure socket connections.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

Related concepts:

“Secure socket API error code messages” on page 51

To get the error code messages for the secure socket API, follow these steps.

Example: GSKit secure server with asynchronous data receive

This example demonstrates how to establish a secure server using Global Security Kit (GSKit) APIs.

The server opens the socket, prepares the secure environment, accepts and processes connection requests, exchanges data with the client and ends the session. The client also opens a socket, sets up the secure environment, calls the server and requests a secure connection, exchanges data with the server, and closes the session. The following diagram and description shows the server/client flow of events.

Note: The following example programs use AF_INET6 address family.

Socket flow of events: Secure server that uses asynchronous data receive



The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between the server and client examples.

1. The `QsoCreateIOCompletionPort()` API creates an I/O completion port.
2. The `pthread_create` API creates a worker thread to receive data and to echo it back to the client. The worker thread waits for client requests to arrive on the I/O completion port just created.
3. A call to `gsk_environment_open()` to obtain a handle to an SSL environment.
4. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the SSL environment. At a minimum, either a call to `gsk_attribute_set_buffer()` to set the `GSK_OS400_APPLICATION_ID` value or to set the `GSK_KEYRING_FILE` value. Only one of these should be set. It is preferred that you use the `GSK_OS400_APPLICATION_ID` value. Also ensure you set the type of application (client or server), `GSK_SESSION_TYPE`, using `gsk_attribute_set_enum()`.
5. A call to `gsk_environment_init()` to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that run using this environment.
6. The socket API creates a socket descriptor. The server then issues the standard set of socket calls: `bind()`, `listen()`, and `accept()` to enable a server to accept incoming connection requests.
7. The `gsk_secure_soc_open()` API obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related API calls.
8. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the secure session. At a minimum, a call to `gsk_attribute_set_numeric_value()` to associate a specific socket with this secure session.
9. A call to `gsk_secure_soc_init()` to initiate the SSL handshake negotiation of the cryptographic parameters.

Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server, to which the client is connecting, has enabled client authentication. The `gsk_attribute_set_buffer(GSK_OS400_APPLICATION_ID)` or `gsk_attribute_set_buffer(GSK_KEYRING_FILE)` API calls identify (though in dissimilar ways) the key database file, from which the certificate and private key that are used during the handshake are obtained.

10. The `gsk_secure_soc_startRecv()` API initiates an asynchronous receive operation on a secure session.
11. The `pthread_join` synchronizes the server and worker programs. This API waits for the thread to end, detaches the thread, and then returns the thread's exit status to the server.
12. The `gsk_secure_soc_close()` API ends the secure session.
13. The `gsk_environment_close()` API closes the SSL environment.
14. The `close()` API ends the listening socket.
15. The `close()` ends the accepted (client connection) socket.
16. The `QsoDestroyIOCompletionPort()` API destroys the completion port.

Socket flow of events: Worker thread that uses GSKit APIs

1. After the server application creates a worker thread, it waits for server to send it the incoming client request to process client data with the `gsk_secure_soc_startRecv()` call. The `QsoWaitForIOCompletionPort()` API waits on the supplied I/O completion port that was specified by the server.
2. As soon as the client request has been received, the `gsk_attribute_get_numeric_value()` API gets the socket descriptor associated with the secure session.
3. The `gsk_secure_soc_write()` API sends the message to the client using the secure session.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* GSK Asynchronous Server Program using ApplicationId*/

/* "IBM grants you a nonexclusive copyright license */
/* to use all programming code examples, from which */
/* you can generate similar function tailored to your */
/* own specific needs. */
/* */
/* All sample code is provided by IBM for illustrative*/
/* purposes only. These examples have not been */
/* thoroughly tested under all conditions. IBM, */
/* therefore, cannot guarantee or imply reliability, */
/* serviceability, or function of these programs. */
/* */
/* All programs contained herein are provided to you */
/* "AS IS" without any warranties of any kind. The */
/* implied warranties of non-infringement, */
/* merchantability and fitness for a particular */
/* purpose are expressly disclaimed. " */

/* Assumes that application id is already registered */
/* and a certificate has been associated with the */
/* application id. */
/* No parameters, some comments and many hardcoded */
/* values to keep it short and simple */

/* use following command to create bound program: */
/* CRTBNDC PGM(PROG/GSKSERVa) */
/* SRCFILE(PROG/CSRC) */
/* SRCMBR(GSKSERVa) */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define MULTI_THREADED
#include "pthread.h"
#include "qsoasync.h"
#define Failure 0
#define Success 1
#define TRUE 1
#define FALSE 0
void *workerThread(void *arg);
/*****
/* Descriptive Name: Master thread will establish a client */
/* connection and hand processing responsibility */
/* to a worker thread. */
/* Note: Due to the thread attribute of this program, spawn() must */
/* be used to invoke. */
*****/
int main(void)
{
    gsk_handle my_env_handle=NULL; /* secure environment handle */
    gsk_handle my_session_handle=NULL; /* secure session handle */

    struct sockaddr_in6 address;
    int buf_len, on = 1, rc = 0;
    int sd = -1, lsd = -1, al = -1, ioCompPort = -1;
    int successFlag = FALSE;
    char buff[1024];
    pthread_t thr;
```

```

void *status;
Qso_OverlappedIO_t ioStruct;

/*****
/* Issue all of the command in a do/while */
/* loop so that clean up can happen at end */
*****/
do
{
    /*****/
    /* Create an I/O completion port for this */
    /* process. */
    /*****/
    if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
    {
        perror("QsoCreateIOCompletionPort() failed");
        break;
    }
    /*****/
    /* Create a worker thread */
    /* to process all client requests. The */
    /* worker thread will wait for client */
    /* requests to arrive on the I/O completion */
    /* port just created. */
    /*****/
    rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
    if (rc < 0)
    {
        perror("pthread_create() failed");
        break;
    }

    /* open a gsk environment */
    rc = errno = 0;
    rc = gsk_environment_open(&my_env_handle);
    if (rc != GSK_OK)
    {
        printf("gsk_environment_open() failed with rc = %d & errno = %d.\n",
            rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }

    /* set the Application ID to use */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_OS400_APPLICATION_ID,
                                "MY_SERVER_APP",
                                13);
    if (rc != GSK_OK)
    {
        printf("gsk_attribute_set_buffer() failed with rc = %d & errno = %d.\n",
            rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }

    /* set this side as the server */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
                                GSK_SESSION_TYPE,
                                GSK_SERVER_SESSION);
    if (rc != GSK_OK)
    {
        printf("gsk_attribute_set_enum() failed with rc = %d & errno = %d.\n",
            rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    }
}

```

```

    break;
}

/* by default TLSV10, TLSV11, and TLSV12 are enabled */
/* We will disable SSL_V3 for this example.          */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_PROTOCOL_SSLV3,
                           GSK_PROTOCOL_SSLV3_OFF);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* We will disable TLS_V10 for this example.          */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_PROTOCOL_TLSV10,
                           GSK_FALSE);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* set the cipher suite to use. By default our default list */
/* of ciphers is enabled. For this example we will just use one */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                             GSK_TLSV12_CIPHER_SPECS_EX,
                             "TLS_RSA_WITH_AES_128_CBC_SHA",
                             28);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_buffer() failed with rc = %d & errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* Initialize the secure environment */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
    printf("gsk_environment_init() failed with rc = %d & errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* initialize a socket to be used for listening */
lfd = socket(AF_INET6, SOCK_STREAM, 0);
if (lfd < 0)
{
    perror("socket() failed");
    break;
}

/* set socket so can be reused immediately */
rc = setsockopt(lfd, SOL_SOCKET,
               SO_REUSEADDR,

```

```

                (char *)&on,
                sizeof(on));
if (rc < 0)
{
    perror("setsockopt() failed");
    break;
}

/* bind to the local server address */
memset((char *) &address, 0, sizeof(address));
address.sin6_family = AF_INET6;
address.sin6_port = 13333;
memcpy(&address.sin6_addr, &in6addr_any, sizeof(in6addr_any));
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("bind() failed");
    break;
}

/* enable the socket for incoming client connections */
listen(lsd, 5);
if (rc < 0)
{
    perror("listen() failed");
    break;
}

/* accept an incoming client connection */
al = sizeof(address);
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
    perror("accept() failed");
    break;
}

/* open a secure session */
rc = errno = 0;
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
    printf("gsk_secure_soc_open() failed with rc = %d & errno = %d.\n",
           rc, errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* associate our socket with the secure session */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);
if (rc != GSK_OK)
{
    printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
    printf("and errno = %d.\n", errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* initiate the SSL handshake */
rc = errno = 0;
rc = gsk_secure_soc_init(my_session_handle);
if (rc != GSK_OK)
{
    printf("gsk_secure_soc_init() failed with rc = %d & errno = %d.\n",

```

```

        rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}
/*****
/* Issue gsk_secure_soc_startRecv() to
/* receive client request.
/* Note:
/* postFlag == on denoting request should
/* posted to the I/O completion port, even
/* if request is immediately available.
/* Worker thread will process client request.*/
*****/
/*****
/* initialize Qso_OverlappedIO_t structure -
/* reserved fields must be hex 00's.
*****/
memset(&ioStruct, '\0', sizeof(ioStruct));
memset((char *) buff, 0, sizeof(buff));
ioStruct.buffer = buff;
ioStruct.bufferLength = sizeof(buff);

/*****
/* Store the session handle in the
/* Qso_OverlappedIO_t descriptorHandle field.*/
/* This area is used to house information
/* defining the state of the client
/* connection. Field descriptorHandle is
/* defined as a (void *) to allow the server
/* to address more extensive client
/* connection state if needed.
*****/
ioStruct.descriptorHandle = my_session_handle;
ioStruct.postFlag = 1;
ioStruct.fillBuffer = 0;

rc = gsk_secure_soc_startRecv(my_session_handle,
                             ioCompPort,
                             &ioStruct);
if (rc != GSK_AS400_ASYNCHRONOUS_RECV)
{
    printf("gsk_secure_soc_startRecv() rc = %d & errno = %d.\n",rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}
/*****
/* This is where the server can loop back
/* to accept a new connection.
*****/

/*****
/* Wait for worker thread to finish
/* processing client connection.
*****/
rc = pthread_join(thr, &status);

/* check status of the worker */
if ( rc == 0 && (rc = __INT(status)) == Success)
{
    printf("Success.\n");
    successFlag = TRUE;
}
else
{
    perror("pthread_join() reported failure");
}

```



```

} while(FALSE);

/* disable the SSL session */
if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

/* disable the SSL environment */
if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

/* close the listening socket */
if (lfd > -1)
    close(lfd);
/* close the accepted socket */
if (sd > -1)
    close(sd);

/* destroy the completion port */
if (ioCompPort > -1)
    QsoDestroyIOCompletionPort(ioCompPort);

if (successFlag)
    exit(0);
else
    exit(-1);
}
/*****
/* Function Name: workerThread */
/*
/* Descriptive Name: Process client connection.
/*
/* Note: To make the sample more straight forward the main routine */
/* handles all of the clean up although this function can */
/* be made responsible for the clientfd and session_handle. */
*****/
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort = -1, clientfd = -1;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    int amtWritten;
    gsk_handle client_session_handle = NULL;
    pthread_t thr;
    pthread_id_np_t t_id;
    t_id = pthread_getthreadid_np();
    tID = t_id.intId.lo;
    /*****
    /* I/O completion port is passed to this */
    /* routine. */
    *****/
    ioCompPort = *(int *)arg;
    /*****
    /* Wait on the supplied I/O completion port */
    /* for a client request. */
    *****/
    waitTime.tv_sec = 500;
    waitTime.tv_usec = 0;
    rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
    if ((rc == 1) &&
        (ioStruct.returnValue == GSK_OK) &&
        (ioStruct.operationCompleted == GSKSECURESOCSTARTRECV))
    /*****
    /* Client request has been received. */
    *****/
    ;
    else

```

```

{
    perror("QsoWaitForIOCompletion()/gsk_secure_soc_startRecv() failed");
    printf("ioStruct.returnValue = %d.\n", ioStruct.returnValue);
    return __VOID(Failure);
}

/* write results to screen */
printf("gsk_secure_soc_startRecv() received %d bytes, here they are:\n",
       ioStruct.secureDataTransferSize);
printf("%s\n",ioStruct.buffer);

/*****
/* Obtain the session handle associated
/* with the client connection.
*****/
client_session_handle = ioStruct.descriptorHandle;

/* get the socket associated with the secure session */
rc=errno=0;
rc = gsk_attribute_get_numeric_value(client_session_handle,
                                   GSK_FD,
                                   &clientfd);

if (rc != GSK_OK)
{
    printf("gsk_attribute_get_numeric_value() rc = %d & errno = %d.\n",
          rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    return __VOID(Failure);
}

/* send the message to the client using the secure session */
amtWritten = 0;
rc = gsk_secure_soc_write(client_session_handle,
                          ioStruct.buffer,
                          ioStruct.secureDataTransferSize,
                          &amtWritten);
if (amtWritten != ioStruct.secureDataTransferSize)
{
    if (rc != GSK_OK)
    {
        printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",
              rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        return __VOID(Failure);
    }
    else
    {
        printf("gsk_secure_soc_write() did not write all data.\n");
        return __VOID(Failure);
    }
}

/* write results to screen */
printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
printf("%s\n",ioStruct.buffer);

return __VOID(Success);
} /* end workerThread */

```

Related concepts:

“Global Security Kit (GSKit) APIs” on page 47

Global Security Kit (GSKit) is a set of programmable interfaces that allow an application to be SSL enabled.

Related reference:

“Example: Establishing a secure client with Global Security Kit APIs” on page 143

This example demonstrates how to establish a client using the Global Security Kit (GSKit) APIs.

“Example: GSKit secure server with asynchronous handshake”

The `gsk_secure_soc_startInit()` API allows you to create secure server applications that can handle requests asynchronously.

Related information:

`QsoCreateIOCompletionPort()`--Create I/O Completion Port API

`pthread_create`

`QsoWaitForIOCompletion()`--Wait for I/O Operation API

`QsoDestroyIOCompletionPort()`--Destroy I/O Completion Port API

`bind()`--Set Local Address for Socket API

`socket()`--Create Socket API

`listen()`--Invite Incoming Connections Requests API

`close()`--Close File or Socket Descriptor API

`accept()`--Wait for Connection Request and Make Connection API

`gsk_environment_open()`--Get a handle for an SSL environment API

`gsk_attribute_set_buffer()`--Set character information for a secure session or an SSL environment API

`gsk_attribute_set_enum()`--Set enumerated information for a secure session or an SSL environment API

`gsk_environment_init()`--Initialize an SSL environment API

`gsk_secure_soc_open()`--Get a handle for a secure session API

`gsk_attribute_set_numeric_value()`--Set numeric information for a secure session or an SSL environment API

`gsk_secure_soc_init()`--Negotiate a secure session API

`gsk_secure_soc_startRecv()`--Start asynchronous receive operation on a secure session API

`pthread_join`

`gsk_secure_soc_close()`--Close a secure session API

`gsk_environment_close()`--Close an SSL environment API

`gsk_attribute_get_numeric_value()`--Get numeric information about a secure session or an SSL environment API

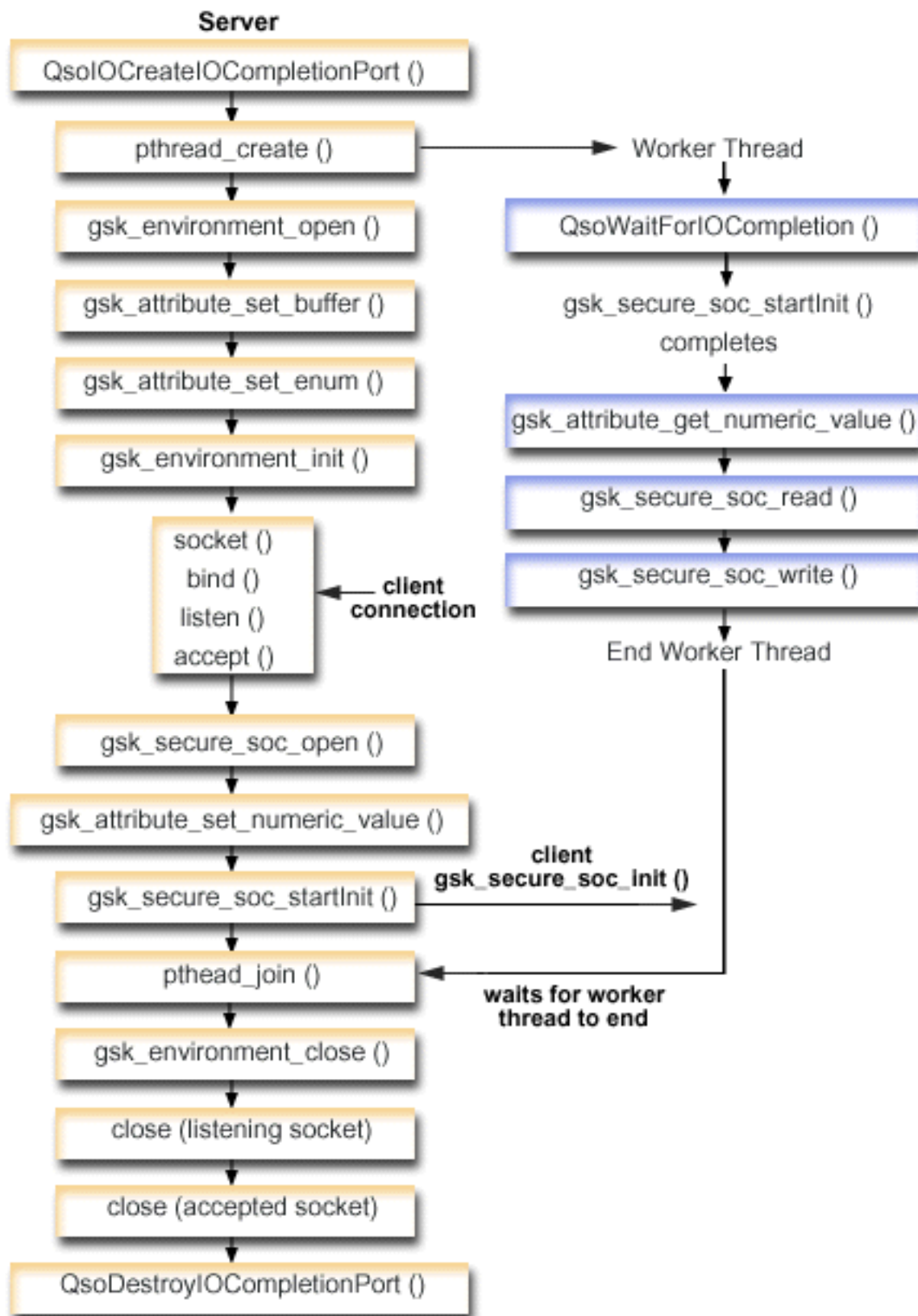
`gsk_secure_soc_write()`--Send data on a secure session API

Example: GSKit secure server with asynchronous handshake

The `gsk_secure_soc_startInit()` API allows you to create secure server applications that can handle requests asynchronously.

The following example illustrates how this API can be used. It is similar to the GSKit secure server with asynchronous data receive example, but uses this API to start a secure session.

The following graphic shows the API calls that are used to negotiate an asynchronous handshake on a secure server.



To view the client portion of this graphic, see GSKit client.

Socket flow of events: GSKit secure server that uses asynchronous handshake

This flow describes the socket calls in the following example application.

1. The `QsoCreateIOCompletionPort()` API creates an I/O completion port.
2. The `pthread_create()` API creates a worker thread to process all client requests. The worker thread waits for client requests to arrive on the I/O completion port just created.
3. A call to `gsk_environment_open()` to obtain a handle to an SSL environment.
4. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the SSL environment. At a minimum, either a call to `gsk_attribute_set_buffer()` to set the `GSK_OS400_APPLICATION_ID` value or to set the `GSK_KEYRING_FILE` value. Only one of these should be set. It is preferred that you use the `GSK_OS400_APPLICATION_ID` value. Also ensure you set the type of application (client or server), `GSK_SESSION_TYPE`, using `gsk_attribute_set_enum()`.
5. A call to `gsk_environment_init()` to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that run using this environment.
6. The socket API creates a socket descriptor. The server then issues the standard set of socket calls, `bind()`, `listen()`, and `accept()`, to enable a server to accept incoming connection requests.
7. The `gsk_secure_soc_open()` API obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related API calls.
8. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the secure session. At a minimum, a call to `gsk_attribute_set_numeric_value()` to associate a specific socket with this secure session.
9. The `gsk_secure_soc_startInit()` API starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session. Control returns to the program here. When the handshake processing is completed, the completion port is posted with the results. The thread can continue on with other processing; however, for simplicity, wait here for the worker thread to complete.

Note: Typically, a server program must provide a certificate for an SSL handshake to succeed. A server must also have access to the private key that is associated with the server certificate and the key database file where the certificate is stored. In some cases, a client must also provide a certificate during the SSL handshake processing. This occurs if the server, which the client is connecting to, has enabled client authentication. The `gsk_attribute_set_buffer(GSK_OS400_APPLICATION_ID)` or `gsk_attribute_set_buffer(GSK_KEYRING_FILE)` API call identifies (though in dissimilar ways) the key database file, from which the certificate and private key that are used during the handshake are obtained.

10. The `pthread_join` synchronizes the server and worker programs. This API waits for the thread to end, detaches the thread, and then returns the thread's exit status to the server.
11. The `gsk_secure_soc_close()` API ends the secure session.
12. The `gsk_environment_close()` API closes the SSL environment.
13. The `close()` API ends the listening socket.
14. The `close()` API ends the accepted (client connection) socket.
15. The `QsoDestroyIOCompletionPort()` API destroys the completion port.

Socket flow of events: Worker thread that process secure asynchronous requests

1. After the server application creates a worker thread, it waits for the server to send it the incoming client request to process. The `QsoWaitForIOCompletionPort()` API waits for the supplied I/O completion port that was specified by the server. This call waits until the `gsk_secure_soc_startInit()` call is completed.
2. As soon as the client request has been received, the `gsk_attribute_get_numeric_value()` API gets the socket descriptor associated with the secure session.

3. The gsk_secure_soc_read() API receives a message from the client using the secure session.
4. The gsk_secure_soc_write() API sends the message to the client using the secure session.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* GSK Asynchronous Server Program using Application Id*/
/* and gsk_secure_soc_startInit() */

/* Assumes that application id is already registered */
/* and a certificate has been associated with the */
/* application id. */
/* No parameters, some comments and many hardcoded */
/* values to keep it short and simple */

/* use following command to create bound program: */
/* CRTBND CPG(MYLIB/GSKSERVSI) */
/* SRCFILE(MYLIB/CSRC) */
/* SRCMBR(GSKSERVSI) */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define _MULTI_THREADED
#include "pthread.h"
#include "qsoasync.h"
#define Failure 0
#define Success 1
#define TRUE 1
#define FALSE 0

void *workerThread(void *arg);
/*****
/* Descriptive Name: Master thread will establish a client */
/* connection and hand processing responsibility */
/* to a worker thread. */
/* Note: Due to the thread attribute of this program, spawn() must */
/* be used to invoke. */
*****/
int main(void)
{
    gsk_handle my_env_handle=NULL; /* secure environment handle */
    gsk_handle my_session_handle=NULL; /* secure session handle */

    struct sockaddr_in6 address;
    int buf_len, on = 1, rc = 0;
    int sd = -1, lsd = -1, al, ioCompPort = -1;
    int successFlag = FALSE;
    pthread_t thr;
    void *status;
    Qso_OverlappedIO_t ioStruct;

    /*****/
    /* Issue all of the command in a do/while */
    /* loop so that clean up can happen at end */
    /*****/

    do
    {
        /*****/
        /* Create an I/O completion port for this */
        /* process. */
        */

```

```

/*****/
if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
{
    perror("QsoCreateIOCompletionPort() failed");
    break;
}
/*****/
/* Create a worker thread */
/* to process all client requests. The */
/* worker thread will wait for client */
/* requests to arrive on the I/O completion */
/* port just created. */
/*****/
rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
if (rc < 0)
{
    perror("pthread_create() failed");
    break;
}

/* open a gsk environment */
rc = errno = 0;
printf("gsk_environment_open()\n");
rc = gsk_environment_open(&my_env_handle);
if (rc != GSK_OK)
{
    printf("gsk_environment_open() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* set the Application ID to use */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_OS400_APPLICATION_ID,
                              "MY_SERVER_APP",
                              13);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* set this side as the server */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_SESSION_TYPE,
                           GSK_SERVER_SESSION);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* by default TLSV10, TLSV11, and TLSV12 are enabled */
/* We will disable SSL_V3 for this example. */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_PROTOCOL_SSLV3,
                           GSK_PROTOCOL_SSLV3_OFF);

if (rc != GSK_OK)
{

```

```

    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* We will disable TLS_V10 for this example.          */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_PROTOCOL_TLSV10,
                           GSK_FALSE);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* set the cipher suite to use. By default our default list
/* of ciphers is enabled. For this example we will just use one */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                             GSK_TLSV12_CIPHER_SPECS_EX,
                             "TLS_RSA_WITH_AES_128_CBC_SHA",
                             28);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* Initialize the secure environment */
rc = errno = 0;
printf("gsk_environment_init()\n");
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
    printf("gsk_environment_init() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* initialize a socket to be used for listening */
printf("socket()\n");
lfd = socket(AF_INET6, SOCK_STREAM, 0);
if (lfd < 0)
{
    perror("socket() failed");
    break;
}

/* set socket so can be reused immediately */
rc = setsockopt(lfd, SOL_SOCKET,
               SO_REUSEADDR,
               (char *)&on,
               sizeof(on));

if (rc < 0)
{
    perror("setsockopt() failed");
    break;
}

/* bind to the local server address */

```



```

memset((char *) &address, 0, sizeof(address));
address.sin6_family = AF_INET6;
address.sin6_port = 13333;
memcpy(&address.sin6_addr, &in6addr_any, sizeof(in6addr_any));
printf("bind()\n");
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("bind() failed");
    break;
}

/* enable the socket for incoming client connections */
printf("listen()\n");
listen(lsd, 5);
if (rc < 0)
{
    perror("listen() failed");
    break;
}

/* accept an incoming client connection */
al = sizeof(address);
printf("accept()\n");
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
    perror("accept() failed");
    break;
}

/* open a secure session */
rc = errno = 0;
printf("gsk_secure_soc_open()\n");
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
    printf("gsk_secure_soc_open() failed with rc = %d and errno = %d.\n",
           rc, errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* associate our socket with the secure session */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
    printf("and errno = %d.\n", errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/*****/
/* Issue gsk_secure_soc_startInit() to */
/* process SSL Handshake flow asynchronously */
/*****/
/*****/
/* initialize Qso_OverlappedIO_t structure - */
/* reserved fields must be hex 00's. */
/*****/
memset(&ioStruct, '\0', sizeof(ioStruct));

/*****/
/* Store the session handle in the */

```

```

/* Qso_OverlappedIO_t descriptorHandle field.*/
/* This area is used to house information */
/* defining the state of the client */
/* connection. Field descriptorHandle is */
/* defined as a (void *) to allow the server */
/* to address more extensive client */
/* connection state if needed. */
/*******/
ioStruct.descriptorHandle = my_session_handle;

/* initiate the SSL handshake */
rc = errno = 0;
printf("gsk_secure_soc_startInit()\n");
rc = gsk_secure_soc_startInit(my_session_handle, ioCompPort, &ioStruct);
if (rc != GSK_OS400_ASYNCHRONOUS_SOC_INIT)
{
    printf("gsk_secure_soc_startInit() rc = %d and errno = %d.\n",rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}
else
    printf("gsk_secure_soc_startInit got GSK_OS400_ASYNCHRONOUS_SOC_INIT\n");

/*******/
/* This is where the server can loop back */
/* to accept a new connection. */
/*******/

/*******/
/* Wait for worker thread to finish */
/* processing client connection. */
/*******/
rc = pthread_join(thr, &status);

/* check status of the worker */
if ( rc == 0 && (rc == __INT(status)) == Success)
{
    printf("Success.\n");
    successFlag = TRUE;
}
else
{
    perror("pthread_join() reported failure");
}
} while(FALSE);

/* disable the SSL session */
if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

/* disable the SSL environment */
if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

/* close the listening socket */
if (lfd > -1)
    close(lfd);
/* close the accepted socket */
if (sd > -1)
    close(sd);

/* destroy the completion port */
if (ioCompPort > -1)
    QsoDestroyIOCompletionPort(ioCompPort);

if (successFlag)

```

```

        exit(0);

    exit(-1);
}

/*****
/* Function Name: workerThread */
/* */
/* Descriptive Name: Process client connection. */
/* */
/* Note: To make the sample more straight forward the main routine */
/*       handles all of the clean up although this function can */
/*       be made responsible for the clientfd and session_handle. */
*****/
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort, clientfd;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    int amtWritten, amtRead;
    char buff[1024];
    gsk_handle client_session_handle;
    pthread_t thr;
    pthread_id_np_t t_id;
    t_id = pthread_getthreadid_np();
    tID = t_id.intId.lo;
    /*****
    /* I/O completion port is passed to this */
    /* routine. */
    *****/
    ioCompPort = *(int *)arg;
    /*****
    /* Wait on the supplied I/O completion port */
    /* for the SSL handshake to complete. */
    *****/
    waitTime.tv_sec = 500;
    waitTime.tv_usec = 0;

    sleep(4);
    printf("QsoWaitForIOCompletion()\n");
    rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
    if ((rc == 1) &&
        (ioStruct.returnValue == GSK_OK) &&
        (ioStruct.operationCompleted == GSKSECURESOCSTARTINIT))
    /*****
    /* SSL Handshake has completed. */
    *****/
    ;
    else
    {
        printf("QsoWaitForIOCompletion()/gsk_secure_soc_startInit() failed.\n");
        printf("rc == %d, returnValue = %d, operationCompleted = %d\n",
            rc, ioStruct.returnValue, ioStruct.operationCompleted);
        perror("QsoWaitForIOCompletion()/gsk_secure_soc_startInit() failed");
        return __VOID(Failure);
    }

    /*****
    /* Obtain the session handle associated */
    /* with the client connection. */
    *****/
    client_session_handle = ioStruct.descriptorHandle;

    /* get the socket associated with the secure session */
    rc=errno=0;

```

```

printf("gsk_attribute_get_numeric_value()\n");
rc = gsk_attribute_get_numeric_value(client_session_handle,
                                     GSK_FD,
                                     &clientfd);

if (rc != GSK_OK)
{
    printf("gsk_attribute_get_numeric_value() rc = %d and errno = %d.\n",
          rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    return __VOID(Failure);
}
/* memset buffer to hex zeros */
memset((char *) buff, 0, sizeof(buff));
amtRead = 0;
/* receive a message from the client using the secure session */
printf("gsk_secure_soc_read()\n");
rc = gsk_secure_soc_read(client_session_handle,
                          buff,
                          sizeof(buff),
                          &amtRead);

if (rc != GSK_OK)
{
    printf("gsk_secure_soc_read() rc = %d and errno = %d.\n",rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    return;
}

/* write results to screen */
printf("gsk_secure_soc_read() received %d bytes, here they are ...\n",
      amtRead);
printf("%s\n",buff);

/* send the message to the client using the secure session */
amtWritten = 0;
printf("gsk_secure_soc_write()\n");
rc = gsk_secure_soc_write(client_session_handle,
                          buff,
                          amtRead,
                          &amtWritten);

if (amtWritten != amtRead)
{
    if (rc != GSK_OK)
    {
        printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        return __VOID(Failure);
    }
    else
    {
        printf("gsk_secure_soc_write() did not write all data.\n");
        return __VOID(Failure);
    }
}
/* write results to screen */
printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
printf("%s\n",buff);

return __VOID(Success);
}
/* end workerThread */

```

Related concepts:

“Global Security Kit (GSKit) APIs” on page 47

Global Security Kit (GSKit) is a set of programmable interfaces that allow an application to be SSL enabled.

Related reference:

“Example: Establishing a secure client with Global Security Kit APIs”

This example demonstrates how to establish a client using the Global Security Kit (GSKit) APIs.

“Example: GSKit secure server with asynchronous data receive” on page 123

This example demonstrates how to establish a secure server using Global Security Kit (GSKit) APIs.

Related information:

QsoCreateIOCompletionPort()--Create I/O Completion Port API

pthread_create

QsoWaitForIOCompletion()--Wait for I/O Operation API

QsoDestroyIOCompletionPort()--Destroy I/O Completion Port API

bind()--Set Local Address for Socket API

socket()--Create Socket API

listen()--Invite Incoming Connections Requests API

close()--Close File or Socket Descriptor API

accept()--Wait for Connection Request and Make Connection API

gsk_environment_open()--Get a handle for an SSL environment API

gsk_attribute_set_buffer()--Set character information for a secure session or an SSL environment API

gsk_attribute_set_enum()--Set enumerated information for a secure session or an SSL environment API

gsk_environment_init()--Initialize an SSL environment API

gsk_secure_soc_open()--Get a handle for a secure session API

gsk_attribute_set_numeric_value()--Set numeric information for a secure session or an SSL environment API

gsk_secure_soc_init()--Negotiate a secure session API

pthread_join

gsk_secure_soc_close()--Close a secure session API

gsk_environment_close()--Close an SSL environment API

gsk_attribute_get_numeric_value()--Get numeric information about a secure session or an SSL environment API

gsk_secure_soc_write()--Send data on a secure session API

gsk_secure_soc_startInit()--Start asynchronous operation to negotiate a secure session API

gsk_secure_soc_read()--Receive data on a secure session API

Example: Establishing a secure client with Global Security Kit APIs

This example demonstrates how to establish a client using the Global Security Kit (GSKit) APIs.

The following graphic shows the API calls on a secure client using the GSKit APIs.



Socket flow of events: GSKit client

This flow describes the socket calls in the following sample application. Use this client example with the GSKit server example and the Example: GSKit secure server with asynchronous handshake.

1. The `gsk_environment_open()` API obtains a handle to an SSL environment.
2. One or more calls to `gsk_attribute_set_xxxx()` to set attributes of the SSL environment. At a minimum, either a call to `gsk_attribute_set_buffer()` to set the `GSK_OS400_APPLICATION_ID` value or to set the `GSK_KEYRING_FILE` value. Only one of these should be set. It is preferred that you use the `GSK_OS400_APPLICATION_ID` value. Also ensure you set the type of application (client or server), `GSK_SESSION_TYPE`, using `gsk_attribute_set_enum()`.
3. A call to `gsk_environment_init()` to initialize this environment for SSL processing and to establish the SSL security information for all SSL sessions that run using this environment.
4. The `socket()` API creates a socket descriptor. The client then issues the `connect()` API to connect to the server application.
5. The `gsk_secure_soc_open()` API obtains storage for a secure session, sets default values for attributes, and returns a handle that must be saved and used on secure session-related API calls.
6. The `gsk_attribute_set_numeric_value()` API associates a specific socket with this secure session.
7. The `gsk_secure_soc_init()` API starts an asynchronous negotiation of a secure session, using the attributes set for the SSL environment and the secure session.
8. The `gsk_secure_soc_write()` API writes data on a secure session to the worker thread.

Note: For the GSKit server example, this API writes data to the worker thread where the `gsk_secure_soc_startRecv()` API is completed. In the asynchronous example, it writes to the completed `gsk_secure_soc_startInit()`.

9. The `gsk_secure_soc_read()` API receives a message from the worker thread using the secure session.
10. The `gsk_secure_soc_close()` API ends the secure session.
11. The `gsk_environment_close()` API closes the SSL environment.
12. The `close()` API ends the connection.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* GSK Client Program using Application Id          */
/*
/* This program assumes that the application id is  */
/* already registered and a certificate has been    */
/* associated with the application id               */
/*
/* No parameters, some comments and many hardcoded */
/* values to keep it short and simple              */
/*
/* use following command to create bound program:  */
/* CRTBNDC PGM(MYLIB/GSKCLIENT)                   */
/*          SRCFILE(MYLIB/CSRC)                     */
/*          SRCMBR(GSKCLIENT)                      */
/*
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define TRUE 1
#define FALSE 0

void main(void)
{
```

```

gsk_handle my_env_handle=NULL;    /* secure environment handle */
gsk_handle my_session_handle=NULL; /* secure session handle */

struct sockaddr_in6 address;
int buf_len, rc = 0, sd = -1;
int amtWritten, amtRead;
char buff1[1024];
char buff2[1024];

/* hardcoded IP address (change to make address where server program runs) */
char addr[40] = "FE80::1";

/*****
/* Issue all of the command in a do/while */
/* loop so that cleanup can happen at end */
*****/
do
{
    /* open a gsk environment */
    rc = errno = 0;
    rc = gsk_environment_open(&my_env_handle);
    if (rc != GSK_OK)
    {
        printf("gsk_environment_open() failed with rc = %d and errno = %d.\n",
               rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }

    /* set the Application ID to use */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                   GSK_OS400_APPLICATION_ID,
                                   "MY_CLIENT_APP",
                                   13);

    if (rc != GSK_OK)
    {
        printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
               rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }

    /* set this side as the client (this is the default */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
                                   GSK_SESSION_TYPE,
                                   GSK_CLIENT_SESSION);

    if (rc != GSK_OK)
    {
        printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
               rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }

    /* by default TLSV10, TLSV11, and TLSV12 are enabled */
    /* We will disable SSL_V3 for this example. */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
                                   GSK_PROTOCOL_SSLV3,
                                   GSK_PROTOCOL_SSLV3_OFF);

    if (rc != GSK_OK)
    {
        printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
               rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    }
}

```



```

    break;
}

/* We will disable TLS_V10 for this example.          */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_PROTOCOL_TLSV10,
                           GSK_FALSE);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* set the cipher suite to use. By default our default list */
/* of ciphers is enabled. For this example we will just use one */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_TLSV12_CIPHER_SPECS_EX,
                              "TLS_RSA_WITH_AES_128_CBC_SHA",
                              28);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* Initialize the secure environment */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
    printf("gsk_environment_init() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* initialize a socket to be used for listening */
sd = socket(AF_INET6, SOCK_STREAM, 0);
if (sd < 0)
{
    perror("socket() failed");
    break;
}

/* connect to the server using a set port number */
memset((char *) &address, 0, sizeof(address));
address.sin6_family = AF_INET6;
address.sin6_port = 13333;
rc = inet_pton(AF_INET6, addr, &address.sin6_addr.s6_addr);
rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("connect() failed");
    break;
}

/* open a secure session */
rc = errno = 0;
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{

```

```

    printf("gsk_secure_soc_open() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* associate our socket with the secure session */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);

if (rc != GSK_OK)
{
    printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
    printf("and errno = %d.\n", errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* initiate the SSL handshake */
rc = errno = 0;
rc = gsk_secure_soc_init(my_session_handle);
if (rc != GSK_OK)
{
    printf("gsk_secure_soc_init() failed with rc = %d and errno = %d.\n",
           rc,errno);
    printf("rc of %d means %s\n", rc, gsk_strerror(rc));
    break;
}

/* memset buffer to hex zeros */
memset((char *) buff1, 0, sizeof(buff1));

/* send a message to the server using the secure session */
strcpy(buff1,"Test of gsk_secure_soc_write \n\n");

/* send the message to the client using the secure session */
buf_len = strlen(buff1);
amtWritten = 0;
rc = gsk_secure_soc_write(my_session_handle, buff1, buf_len, &amtWritten);
if (amtWritten != buf_len)
{
    if (rc != GSK_OK)
    {
        printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }
    else
    {
        printf("gsk_secure_soc_write() did not write all data.\n");
        break;
    }
}

/* write results to screen */
printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
printf("%s\n",buff1);

/* memset buffer to hex zeros */
memset((char *) buff2, 0x00, sizeof(buff2));

/* receive a message from the client using the secure session */
amtRead = 0;
rc = gsk_secure_soc_read(my_session_handle, buff2, sizeof(buff2), &amtRead);

if (rc != GSK_OK)

```

```

    {
        printf("gsk_secure_soc_read() rc = %d and errno = %d.\n",rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
        break;
    }

    /* write results to screen */
    printf("gsk_secure_soc_read() received %d bytes, here they are ...\n",
        amtRead);
    printf("%s\n",buff2);

} while(FALSE);

/* disable SSL support for the socket */
if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

/* disable the SSL environment */
if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

/* close the connection */
if (sd > -1)
    close(sd);

return;
}

```

Related concepts:

“Global Security Kit (GSKit) APIs” on page 47

Global Security Kit (GSKit) is a set of programmable interfaces that allow an application to be SSL enabled.

Related reference:

“Example: GSKit secure server with asynchronous data receive” on page 123

This example demonstrates how to establish a secure server using Global Security Kit (GSKit) APIs.

“Example: GSKit secure server with asynchronous handshake” on page 133

The `gsk_secure_soc_startInit()` API allows you to create secure server applications that can handle requests asynchronously.

Related information:

`socket()`--Create Socket API

`close()`--Close File or Socket Descriptor API

`connect()`--Establish Connection or Destination Address API

`gsk_environment_open()`--Get a handle for an SSL environment API

`gsk_attribute_set_buffer()`--Set character information for a secure session or an SSL environment API

`gsk_attribute_set_enum()`--Set enumerated information for a secure session or an SSL environment API

`gsk_environment_init()`--Initialize an SSL environment API

`gsk_secure_soc_open()`--Get a handle for a secure session API

`gsk_attribute_set_numeric_value()`--Set numeric information for a secure session or an SSL environment API

`gsk_secure_soc_init()`--Negotiate a secure session API

`gsk_secure_soc_close()`--Close a secure session API

`gsk_environment_close()`--Close an SSL environment API

`gsk_secure_soc_write()`--Send data on a secure session API

`gsk_secure_soc_startInit()`--Start asynchronous operation to negotiate a secure session API

`gsk_secure_soc_startRecv()`--Start asynchronous receive operation on a secure session API

gsk_secure_soc_read()--Receive data on a secure session API

Example: Using gethostbyaddr_r() for threadsafe network routines

This example program uses the gethostbyaddr_r() API. All other routines with names that end in _r have similar semantics and are also threadsafe.

This example program takes an IP address in the dotted-decimal notation and prints the host name.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* Header files
*****/
#include </netdb.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define HEX00 '\x00'
#define NUPARMS 2
/*****
/* Pass one parameter that is the IP address in
/* dotted decimal notation. The host name will be
/* displayed if found; otherwise, a message states
/* host not found.
*****/
int main(int argc, char *argv[])
{
    int rc;
    struct in_addr internet_address;
    struct hostent hst_ent;
    struct hostent_data hst_ent_data;
    char dotted_decimal_address [16];
    char host_name[MAXHOSTNAMELEN];

    /*****
    /* Verify correct number of arguments have been passed
    *****/
    if (argc != NUPARMS)
    {
        printf("Wrong number of parms passed\n");
        exit(-1);
    }

    /*****
    /* Obtain addressability to parameters passed
    *****/
    strcpy(dotted_decimal_address, argv[1]);

    /*****
    /* Initialize the structure-field
    /* hostent_data.host_control_blk with hexadecimal zeros
    /* before its initial use. If you require compatibility
    /* with other platforms, then you must initialize the
    /* entire hostent_data structure with hexadecimal zeros.
    *****/
    /* Initialize to hex 00 hostent_data structure
    *****/
    memset(&hst_ent_data,HEX00,sizeof(struct hostent_data));

    /*****
    /* Translate an IP address from dotted decimal
    /* notation to 32-bit IP address format.
    *****/

```

```

/*****
internet_address.s_addr=inet_addr(dotted_decimal_address);

/*****
/* Obtain host name */
/*****
/*****
/* NOTE: The gethostbyaddr_r() returns an integer. */
/* The following are possible values: */
/* -1 (unsuccessful call) */
/* 0 (successful call) */
/*****
rc=gethostbyaddr_r((char *) &internet_address,
                  sizeof(struct in_addr), AF_INET,
                  &hst_ent, &hst_ent_data);

if (rc== -1)
{
    printf("Host name not found\n");
    exit(-1);
}
else
{
    /*****
    /* Copy the host name to an output buffer */
    /*****
    (void) memcpy((void *) host_name,
    /*****
    /* You must address all the results through the */
    /* hostent structure hst_ent. */
    /* NOTE: Hostent_data structure hst_ent_data is just */
    /* a data repository that is used to support the */
    /* hostent structure. Applications should consider */
    /* hostent_data a storage area to put host level data */
    /* that the application does not need to access. */
    /*****
    (void *) hst_ent.h_name,
    MAXHOSTNAMELEN);

    /*****
    /* Print the host name */
    /*****
    printf("The host name is %s\n", host_name);

    }
    exit(0);
}

```

Related concepts:

“Thread safety” on page 58

A function is considered threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe only if all the functions it calls are also threadsafe. Socket APIs consist of system and network functions, which are both threadsafe.

Related reference:

“Socket network functions” on page 64

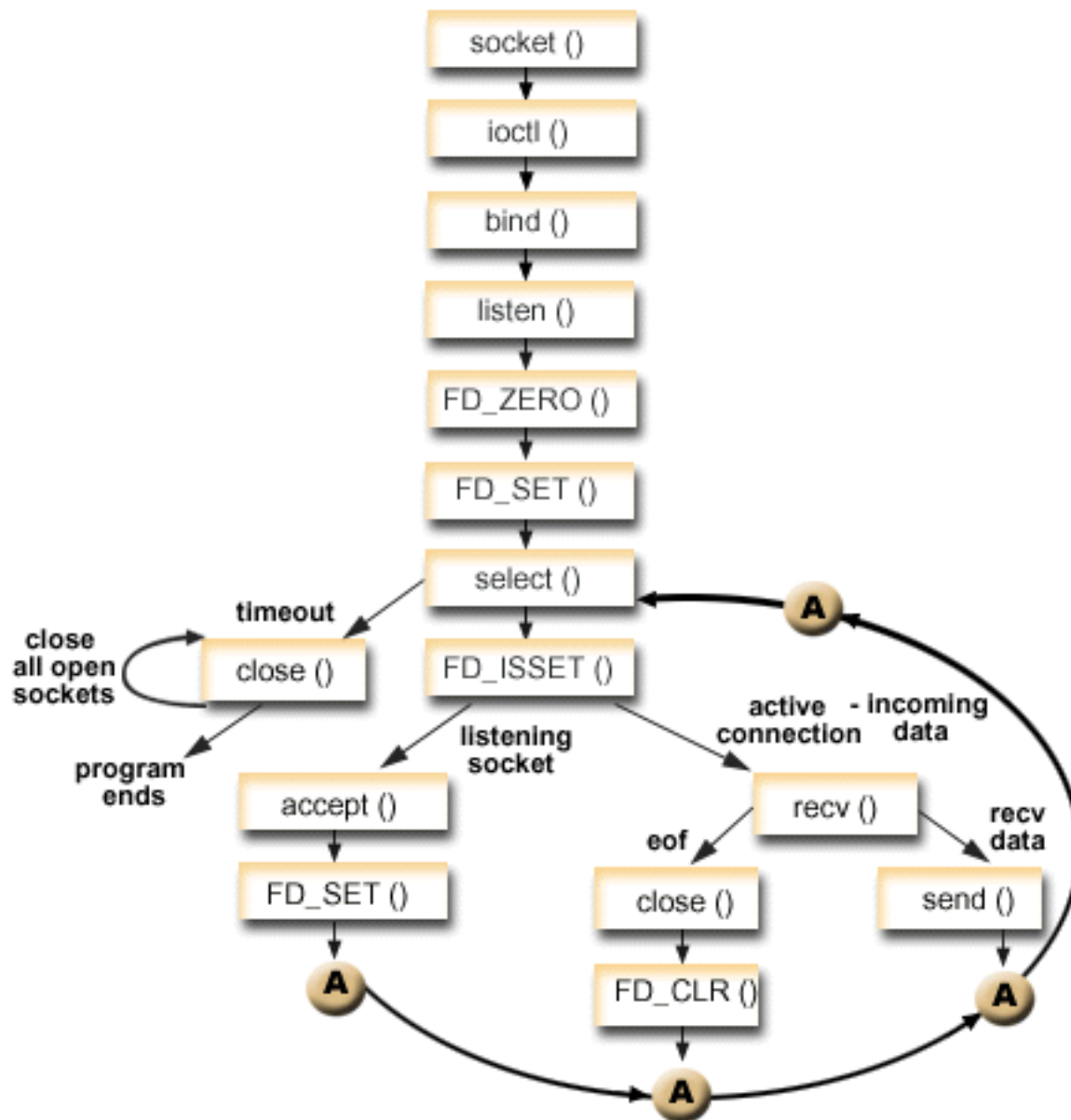
Socket network functions allow application programs to obtain information from the host, protocol, service, and network files.

Related information:

gethostbyaddr_r()--Get Host Information for IP Address API

Example: Nonblocking I/O and select()

This sample program illustrates a server application that uses **nonblocking and the select() API**.



Socket flow of events: Server that uses nonblocking I/O and select()

The following calls are used in the example:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. The `ioctl()` API allows the local address to be reused when the server is restarted before the required wait time expires. In this example, it sets the socket to be nonblocking. All of the sockets for the incoming connections are also nonblocking because they inherit that state from the listening socket.
3. After the socket descriptor is created, the `bind()` gets a unique name for the socket.

4. The listen() allows the server to accept incoming client connections.
5. The server uses the accept() API to accept an incoming connection request. The accept() API call blocks indefinitely, waiting for the incoming connection to arrive.
6. The select() API allows the process to wait for an event to occur and to wake up the process when the event occurs. In this example, the select() API returns a number that represents the socket descriptors that are ready to be processed.
 - 0 Indicates that the process times out. In this example, the timeout is set for 3 minutes.
 - 1 Indicates that the process has failed.
 - 1 Indicates only one descriptor is ready to be processed. In this example, when a 1 is returned, the FD_ISSET and the subsequent socket calls complete only once.
 - n Indicates that multiple descriptors are waiting to be processed. In this example, when an n is returned, the FD_ISSET and subsequent code loops and completes the requests in the order they are received by the server.
7. The accept() and recv() APIs are completed when the EWOULDBLOCK is returned.
8. The send() API echoes the data back to the client.
9. The close() API closes any open socket descriptors.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT 12345

#define TRUE 1
#define FALSE 0

main (int argc, char *argv[])
{
    int i, len, rc, on = 1;
    int listen_sd, max_sd, new_sd;
    int desc_ready, end_server = FALSE;
    int close_conn;
    char buffer[80];
    struct sockaddr_in6 addr;
    struct timeval timeout;
    fd_set master_set, working_set;

    /* Create an AF_INET6 stream socket to receive incoming
     * connections on */
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }

    /* Allow socket descriptor to be reuseable */
    rc = setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));
```

```

if (rc < 0)
{
    perror("setsockopt() failed");
    close(listen_sd);
    exit(-1);
}

/*****
/* Set socket to be nonblocking. All of the sockets for
/* the incoming connections will also be nonblocking since
/* they will inherit that state from the listening socket.
*****/
rc = ioctl(listen_sd, FIONBIO, (char *)&on);
if (rc < 0)
{
    perror("ioctl() failed");
    close(listen_sd);
    exit(-1);
}

/*****
/* Bind the socket
*****/
memset(&addr, 0, sizeof(addr));
addr.sin6_family = AF_INET6;
memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
addr.sin6_port = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
    perror("bind() failed");
    close(listen_sd);
    exit(-1);
}

/*****
/* Set the listen back log
*****/
rc = listen(listen_sd, 32);
if (rc < 0)
{
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
}

/*****
/* Initialize the master fd_set
*****/
FD_ZERO(&master_set);
max_sd = listen_sd;
FD_SET(listen_sd, &master_set);

/*****
/* Initialize the timeval struct to 3 minutes. If no
/* activity after 3 minutes this program will end.
*****/
timeout.tv_sec = 3 * 60;
timeout.tv_usec = 0;

/*****
/* Loop waiting for incoming connects or for incoming data
/* on any of the connected sockets.
*****/
do
{

```



```

/*****
/* Copy the master fd_set over to the working fd_set.    */
/*****
memcpy(&working_set, &master_set, sizeof(master_set));

/*****
/* Call select() and wait 3 minutes for it to complete.  */
/*****
printf("Waiting on select()...\n");
rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

/*****
/* Check to see if the select call failed.                */
/*****
if (rc < 0)
{
    perror(" select() failed");
    break;
}

/*****
/* Check to see if the 3 minute time out expired.         */
/*****
if (rc == 0)
{
    printf(" select() timed out. End program.\n");
    break;
}

/*****
/* One or more descriptors are readable. Need to          */
/* determine which ones they are.                         */
/*****
desc_ready = rc;
for (i=0; i <= max_sd && desc_ready > 0; ++i)
{
    /*****
    /* Check to see if this descriptor is ready           */
    /*****
    if (FD_ISSET(i, &working_set))
    {
        /*****
        /* A descriptor was found that was readable - one  */
        /* less has to be looked for. This is being done   */
        /* so that we can stop looking at the working set  */
        /* once we have found all of the descriptors that  */
        /* were ready.                                     */
        /*****
        desc_ready -= 1;

        /*****
        /* Check to see if this is the listening socket   */
        /*****
        if (i == listen_sd)
        {
            printf(" Listening socket is readable\n");
            /*****
            /* Accept all incoming connections that are    */
            /* queued up on the listening socket before we */
            /* loop back and call select again.             */
            /*****
            do
            {
                /*****
                /* Accept each incoming connection. If     */
                /* accept fails with EWOULDBLOCK, then we   */
                /* have accepted all of them. Any other    */

```

```

/* failure on accept will cause us to end the */
/* server. */
/*****/
new_sd = accept(listen_sd, NULL, NULL);
if (new_sd < 0)
{
    if (errno != EWOULDBLOCK)
    {
        perror(" accept() failed");
        end_server = TRUE;
    }
    break;
}

/*****/
/* Add the new incoming connection to the */
/* master read set */
/*****/
printf(" New incoming connection - %d\n", new_sd);
FD_SET(new_sd, &master_set);
if (new_sd > max_sd)
    max_sd = new_sd;

/*****/
/* Loop back up and accept another incoming */
/* connection */
/*****/
} while (new_sd != -1);
}

/*****/
/* This is not the listening socket, therefore an */
/* existing connection must be readable */
/*****/
else
{
    printf(" Descriptor %d is readable\n", i);
    close_conn = FALSE;
    /*****/
    /* Receive all incoming data on this socket */
    /* before we loop back and call select again. */
    /*****/
    do
    {
        /*****/
        /* Receive data on this connection until the */
        /* recv fails with EWOULDBLOCK. If any other */
        /* failure occurs, we will close the */
        /* connection. */
        /*****/
        rc = recv(i, buffer, sizeof(buffer), 0);
        if (rc < 0)
        {
            if (errno != EWOULDBLOCK)
            {
                perror(" recv() failed");
                close_conn = TRUE;
            }
            break;
        }
    }

    /*****/
    /* Check to see if the connection has been */
    /* closed by the client */
    /*****/
    if (rc == 0)
    {

```

```

        printf(" Connection closed\n");
        close_conn = TRUE;
        break;
    }

    /******
    /* Data was received */
    /******
    len = rc;
    printf(" %d bytes received\n", len);

    /******
    /* Echo the data back to the client */
    /******
    rc = send(i, buffer, len, 0);
    if (rc < 0)
    {
        perror(" send() failed");
        close_conn = TRUE;
        break;
    }

} while (TRUE);

/******
/* If the close_conn flag was turned on, we need */
/* to clean up this active connection. This */
/* clean up process includes removing the */
/* descriptor from the master set and */
/* determining the new maximum descriptor value */
/* based on the bits that are still turned on in */
/* the master set. */
/******
if (close_conn)
{
    close(i);
    FD_CLR(i, &master_set);
    if (i == max_sd)
    {
        while (FD_ISSET(max_sd, &master_set) == FALSE)
            max_sd -= 1;
    }
}
} /* End of existing connection is readable */
} /* End of if (FD_ISSET(i, &working_set)) */
} /* End of loop through selectable descriptors */

} while (end_server == FALSE);

/******
/* Clean up all of the sockets that are open */
/******
for (i=0; i <= max_sd; ++i)
{
    if (FD_ISSET(i, &master_set))
        close(i);
}
}

```

Related concepts:

“Nonblocking I/O” on page 58

When an application issues one of the socket input APIs and there is no data to read, the API blocks and does not return until there is data to read.

“I/O multiplexing—select()” on page 64

Because asynchronous I/O provides a more efficient way to maximize your application resources, it is recommended that you use asynchronous I/O APIs rather than the select() API. However, your specific

application design might allow `select()` to be used.

Related reference:

“Socket application design recommendations” on page 88

Before working with a socket application, assess the functional requirements, goals, and needs of the socket application. Also, consider the performance requirements and the system resource impacts of the application.

“Example: Generic client” on page 113

This example contains the code for a common client job. The client job does a `socket()`, `connect()`, `send()`, `recv()`, and `close()` operation.

Related information:

`accept()`--Wait for Connection Request and Make Connection API

`recv()`--Receive Data API

`ioctl()`--Perform I/O Control Request API

`send()`--Send Data API

`listen()`--Invite Incoming Connections Requests API

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`bind()`--Set Local Address for Socket API

`select()`--Wait for Events on Multiple Sockets API

Using `poll()` instead of `select()`

The `poll()` API is part of the Single Unix Specification and the UNIX 95/98 standard. The `poll()` API performs the same API as the existing `select()` API. The only difference between these two APIs is the interface provided to the caller.

The `select()` API requires that the application pass in an array of bits in which one bit is used to represent each descriptor number. When descriptor numbers are very large, it can overflow the 30KB allocated memory size, forcing multiple iterations of the process. This overhead can adversely affect performance.

The `poll()` API allows the application to pass an array of structures rather than an array of bits. Because each `pollfd` structure can contain up to 8 bytes, the application only needs to pass one structure for each descriptor, even if descriptor numbers are very large.

Socket flow of events: Server that uses `poll()`

The following calls are used in the example:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the `AF_INET6` (Internet Protocol version 6) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. The `setsockopt()` API allows the application to reuse the local address when the server is restarted before the required wait time expires.
3. The `ioctl()` API sets the socket to be nonblocking. All of the sockets for the incoming connections are also nonblocking because they inherit that state from the listening socket.
4. After the socket descriptor is created, the `bind()` API gets a unique name for the socket.
5. The `listen()` API call allows the server to accept incoming client connections.
6. The `poll()` API allows the process to wait for an event to occur and to wake up the process when the event occurs. The `poll()` API might return one of the following values.
 - 0 Indicates that the process times out. In this example, the timeout is set for 3 minutes (in milliseconds).
 - 1 Indicates that the process has failed.

- 1 Indicates only one descriptor is ready to be processed, which is processed only if it is the listening socket.
 - 1++ Indicates that multiple descriptors are waiting to be processed. The poll() API allows simultaneous connection with all descriptors in the queue on the listening socket.
7. The accept() and recv() APIs are completed when the EWOULDBLOCK is returned.
 8. The send() API echoes the data back to the client.
 9. The close() API closes any open socket descriptors.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/poll.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT 12345

#define TRUE 1
#define FALSE 0

main (int argc, char *argv[])
{
    int len, rc, on = 1;
    int listen_sd = -1, new_sd = -1;
    int desc_ready, end_server = FALSE, compress_array = FALSE;
    int close_conn;
    char buffer[80];
    struct sockaddr_in6 addr;
    int timeout;
    struct pollfd fds[200];
    int nfds = 1, current_size = 0, i, j;

    /******
    /* Create an AF_INET6 stream socket to receive incoming */
    /* connections on */
    /******
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }

    /******
    /* Allow socket descriptor to be reuseable */
    /******
    rc = setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));
    if (rc < 0)
    {
        perror("setsockopt() failed");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Set socket to be nonblocking. All of the sockets for */
    /* the incoming connections will also be nonblocking since */
    /* they will inherit that state from the listening socket. */
```

```

/*****/
rc = ioctl(listen_sd, FIONBIO, (char *)&on);
if (rc < 0)
{
    perror("ioctl() failed");
    close(listen_sd);
    exit(-1);
}

/*****/
/* Bind the socket */
/*****/
memset(&addr, 0, sizeof(addr));
addr.sin6_family = AF_INET6;
memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
addr.sin6_port = htons(SERVER_PORT);
rc = bind(listen_sd,
           (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
    perror("bind() failed");
    close(listen_sd);
    exit(-1);
}

/*****/
/* Set the listen back log */
/*****/
rc = listen(listen_sd, 32);
if (rc < 0)
{
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
}

/*****/
/* Initialize the pollfd structure */
/*****/
memset(fds, 0, sizeof(fds));

/*****/
/* Set up the initial listening socket */
/*****/
fds[0].fd = listen_sd;
fds[0].events = POLLIN;

/*****/
/* Initialize the timeout to 3 minutes. If no */
/* activity after 3 minutes this program will end. */
/* timeout value is based on milliseconds. */
/*****/
timeout = (3 * 60 * 1000);

/*****/
/* Loop waiting for incoming connects or for incoming data */
/* on any of the connected sockets. */
/*****/
do
{
    /*****/
    /* Call poll() and wait 3 minutes for it to complete. */
    /*****/
    printf("Waiting on poll()...\n");
    rc = poll(fds, nfd, timeout);

    /*****/
    /* Check to see if the poll call failed. */
    /*****/

```

```

/*****/
if (rc < 0)
{
    perror(" poll() failed");
    break;
}

/*****/
/* Check to see if the 3 minute time out expired.      */
/*****/
if (rc == 0)
{
    printf(" poll() timed out. End program.\n");
    break;
}

/*****/
/* One or more descriptors are readable. Need to      */
/* determine which ones they are.                      */
/*****/
current_size = nfds;
for (i = 0; i < current_size; i++)
{
    /*****/
    /* Loop through to find the descriptors that returned */
    /* POLLIN and determine whether it's the listening    */
    /* or the active connection.                          */
    /*****/
    if(fds[i].revents == 0)
        continue;

    /*****/
    /* If revents is not POLLIN, it's an unexpected result, */
    /* log and end the server.                               */
    /*****/
    if(fds[i].revents != POLLIN)
    {
        printf(" Error! revents = %d\n", fds[i].revents);
        end_server = TRUE;
        break;
    }
    if (fds[i].fd == listen_sd)
    {
        /*****/
        /* Listening descriptor is readable.                */
        /*****/
        printf(" Listening socket is readable\n");

        /*****/
        /* Accept all incoming connections that are      */
        /* queued up on the listening socket before we    */
        /* loop back and call poll again.                  */
        /*****/
        do
        {
            /*****/
            /* Accept each incoming connection. If        */
            /* accept fails with EWOULDBLOCK, then we      */
            /* have accepted all of them. Any other        */
            /* failure on accept will cause us to end the  */
            /* server.                                       */
            /*****/
            new_sd = accept(listen_sd, NULL, NULL);
            if (new_sd < 0)
            {

```

```

        if (errno != EWOULDBLOCK)
        {
            perror(" accept() failed");
            end_server = TRUE;
        }
        break;
    }

    /******
    /* Add the new incoming connection to the          */
    /* pollfd structure                                */
    /******
    printf(" New incoming connection - %d\n", new_sd);
    fds[nfds].fd = new_sd;
    fds[nfds].events = POLLIN;
    nfds++;

    /******
    /* Loop back up and accept another incoming        */
    /* connection                                        */
    /******
    } while (new_sd != -1);
}

/******
/* This is not the listening socket, therefore an      */
/* existing connection must be readable                */
/******

else
{
    printf(" Descriptor %d is readable\n", fds[i].fd);
    close_conn = FALSE;
    /******
    /* Receive all incoming data on this socket        */
    /* before we loop back and call poll again.        */
    /******

do
{
    /******
    /* Receive data on this connection until the      */
    /* recv fails with EWOULDBLOCK. If any other      */
    /* failure occurs, we will close the              */
    /* connection.                                    */
    /******
    rc = recv(fds[i].fd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
        if (errno != EWOULDBLOCK)
        {
            perror(" recv() failed");
            close_conn = TRUE;
        }
        break;
    }

    /******
    /* Check to see if the connection has been        */
    /* closed by the client                            */
    /******
    if (rc == 0)
    {
        printf(" Connection closed\n");
        close_conn = TRUE;
        break;
    }
}

```



```

/*****
/* Data was received */
/*****
len = rc;
printf(" %d bytes received\n", len);

/*****
/* Echo the data back to the client */
/*****
rc = send(fds[i].fd, buffer, len, 0);
if (rc < 0)
{
    perror(" send() failed");
    close_conn = TRUE;
    break;
}

} while(TRUE);

/*****
/* If the close_conn flag was turned on, we need */
/* to clean up this active connection. This */
/* clean up process includes removing the */
/* descriptor. */
/*****
if (close_conn)
{
    close(fds[i].fd);
    fds[i].fd = -1;
    compress_array = TRUE;
}

} /* End of existing connection is readable */
} /* End of loop through pollable descriptors */

/*****
/* If the compress_array flag was turned on, we need */
/* to squeeze together the array and decrement the number */
/* of file descriptors. We do not need to move back the */
/* events and revents fields because the events will always */
/* be POLLIN in this case, and revents is output. */
/*****
if (compress_array)
{
    compress_array = FALSE;
    for (i = 0; i < nfds; i++)
    {
        if (fds[i].fd == -1)
        {
            for(j = i; j < nfds; j++)
            {
                fds[j].fd = fds[j+1].fd;
            }
            i--;
            nfds--;
        }
    }
}

} while (end_server == FALSE); /* End of serving running. */

/*****
/* Clean up all of the sockets that are open */
/*****
for (i = 0; i < nfds; i++)

```

```

{
    if(fds[i].fd >= 0)
        close(fds[i].fd);
}
}

```

Related information:

accept()--Wait for Connection Request and Make Connection API

recv()--Receive Data API

ioctl()--Perform I/O Control Request API

send()--Send Data API

listen()--Invite Incoming Connections Requests API

close()--Close File or Socket Descriptor API

socket()--Create Socket API

bind()--Set Local Address for Socket API

setsockopt()--Set Socket Options API

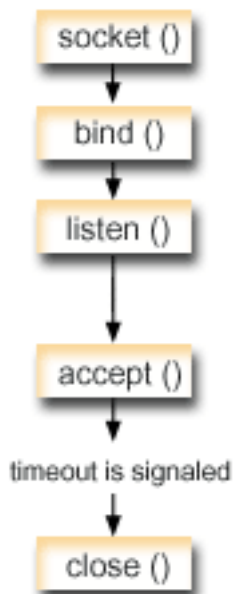
poll()--Wait for Events on Multiple Descriptors API

Example: Using signals with blocking socket APIs

When a process or an application becomes blocked, signals allow you to be notified. They also provide a time limit for blocking processes.

In this example, the signal occurs after five seconds on the accept() call. This call normally blocks indefinitely, but because there is an alarm set, the call blocks only for five seconds. Because blocked programs can hinder performance of an application or a server, you can use signals to diminish this impact. The following example shows how to use signals with blocking socket APIs.

Note: Asynchronous I/O used in a threaded server model is preferable over the more conventional model.



Socket flow of events: Using signals with blocking socket

The following sequence of API calls shows how you can use signals to alert the application when the socket has been inactive:

1. The `socket()` API returns a socket descriptor, which represents an endpoint. The statement also identifies that the `AF_INET6` (Internet Protocol version 6) address family with the TCP transport (`SOCK_STREAM`) is used for this socket.
2. After the socket descriptor is created, a `bind()` API gets a unique name for the socket. In this example, a port number is not specified because the client application does not connect to this socket. This code snippet can be used within other server programs that use blocking APIs, such as `accept()`.
3. The `listen()` API indicates a willingness to accept client connection requests. After the `listen()` API is issued, an alarm is set to go off in five seconds. This alarm or signal alerts you when the `accept()` call blocks.
4. The `accept()` API accepts a client connection request. This call normally blocks indefinitely, but because there is an alarm set, the call only blocks for five seconds. When the alarm goes off, the `accept` call is completed with -1 and with an `errno` value of `EINTR`.
5. The `close()` API ends any open socket descriptors.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* Example shows how to set alarms for blocking socket APIs */

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>

/* Signal catcher routine. This routine will be called when the
 * signal occurs. */
void catcher(int sig)
{
    printf("    Signal catcher called for signal %d\n", sig);
}

/* Main program */
int main(int argc, char *argv[])
{
    struct sigaction sact;
    struct sockaddr_in6 addr;
    time_t t;
    int sd, rc;

    /* Create an AF_INET6, SOCK_STREAM socket */
    printf("Create a TCP socket\n");
    sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (sd == -1)
    {
```

```

        perror("    socket failed");
        return(-1);
    }

/*****
/* Bind the socket.  A port number was not specified because
/* we are not going to ever connect to this socket.
*****/
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family = AF_INET6;
    printf("Bind the socket\n");
    rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
    if (rc != 0)
    {
        perror("    bind failed");
        close(sd);
        return(-2);
    }

/*****
/* Perform a listen on the socket.
*****/
    printf("Set the listen backlog\n");
    rc = listen(sd, 5);
    if (rc != 0)
    {
        perror("    listen failed");
        close(sd);
        return(-3);
    }

/*****
/* Set up an alarm that will go off in 5 seconds.
*****/
    printf("\nSet an alarm to go off in 5 seconds. This alarm will cause the\n");
    printf("blocked accept() to return a -1 and an errno value of EINTR.\n\n");
    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGALRM, &sact, NULL);
    alarm(5);

/*****
/* Display the current time when the alarm was set
*****/
    time(&t);
    printf("Before accept(), time is %s", ctime(&t));

/*****
/* Call accept.  This call will normally block indefinitely,
/* but because we have an alarm set, it will only block for
/* 5 seconds.  When the alarm goes off, the accept call will
/* complete with -1 and an errno value of EINTR.
*****/
    errno = 0;
    printf("    Wait for an incoming connection to arrive\n");
    rc = accept(sd, NULL, NULL);
    printf("    accept() completed.  rc = %d, errno = %d\n", rc, errno);
    if (rc >= 0)
    {
        printf("    Incoming connection was received\n");
        close(rc);
    }
    else
    {
        perror("    errno string");
    }
}

```

```

/*****
/* Show what time it was when the alarm went off          */
*****/
    time(&t);
    printf("After accept(), time is %s\n", ctime(&t));
    close(sd);
    return(0);
}

```

Related concepts:

“Signals” on page 60

An application program can request to be notified asynchronously (request that the system send a *signal*) when a condition that the application is interested in occurs.

“Asynchronous I/O” on page 43

Asynchronous I/O APIs provide a method for threaded client/server models to perform highly concurrent and memory-efficient I/O.

Related reference:

“Berkeley Software Distribution compatibility” on page 68

Sockets is a Berkeley Software Distribution (BSD) interface.

“Example: Using asynchronous I/O” on page 116

An application creates an I/O completion port using the `QsoCreateIOCompletionPort()` API. This API returns a handle that can be used to schedule and wait for completion of asynchronous I/O requests.

Related information:

`accept()`--Wait for Connection Request and Make Connection API

`listen()`--Invite Incoming Connections Requests API

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`bind()`--Set Local Address for Socket API

Examples: Using multicasting with AF_INET

With IP multicasting, an application can send a single IP datagram that a group of hosts in a network can receive.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

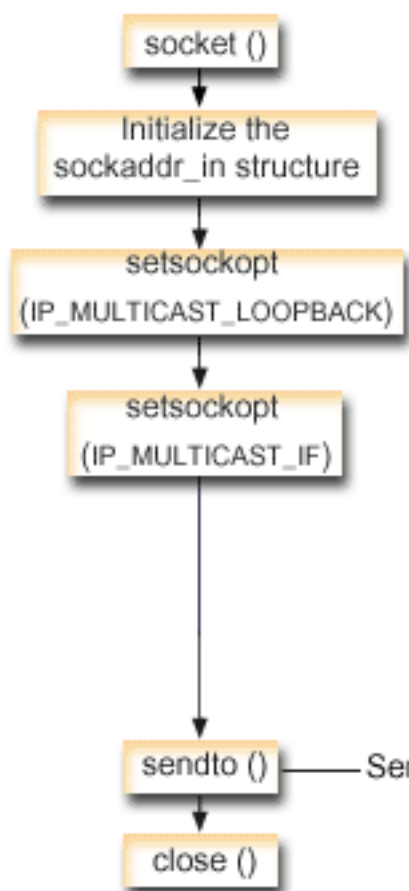
The hosts that are in the group might reside on a single subnet or on different subnets that connect multicast-capable routers. Hosts might join and leave groups at any time. There are no restrictions on the location or number of members in a host group. A class D IP address in the range 224.0.0.1 to 239.255.255.255 identifies a host group.

An application program can send or receive multicast datagrams by using the `socket()` API and connectionless `SOCK_DGRAM` type sockets. Multicasting is a one-to-many transmission method. You cannot use connection-oriented sockets of type `SOCK_STREAM` for multicasting. When a socket of type `SOCK_DGRAM` is created, an application can use the `setsockopt()` API to control the multicast characteristics associated with that socket. The `setsockopt()` API accepts the following `IPPROTO_IP` level flags:

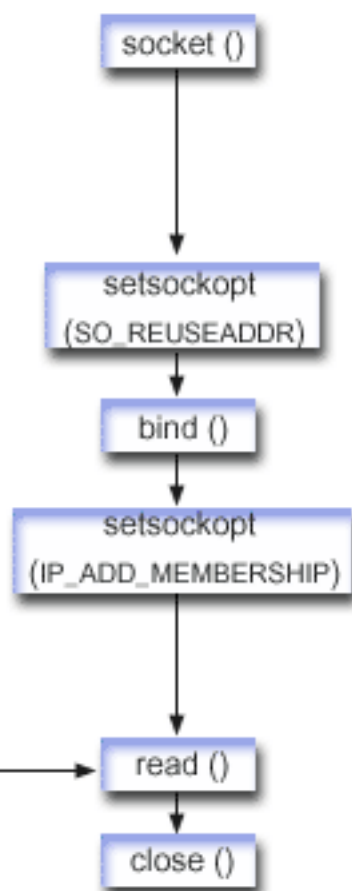
- `IP_ADD_MEMBERSHIP`: Joins the multicast group specified.
- `IP_DROP_MEMBERSHIP`: Leaves the multicast group specified.
- `IP_MULTICAST_IF`: Sets the interface over which outgoing multicast datagrams are sent.
- `IP_MULTICAST_TTL`: Sets the Time To Live (TTL) in the IP header for outgoing multicast datagrams.
- `IP_MULTICAST_LOOP`: Specifies whether a copy of an outgoing multicast datagram is delivered to the sending host as long as it is a member of the multicast group.

Note: IBM i sockets support IP multicasting for the AF_INET address family.

Sending multicast datagrams



Receiving multicast datagrams



Sends datagrams

Socket flow of events: Sending multicast datagrams

The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between two applications that send and receive multicast datagrams. The first example uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor representing an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (SOCK_DGRAM) is used for this socket. This socket sends datagrams to another application.
2. The `sockaddr_in` structure specifies the destination IP address and port number. In this example, the address is 225.1.1.1 and the port number is 5555.
3. The `setsockopt()` API sets the `IP_MULTICAST_LOOP` socket option so that the sending system does not receive a copy of the multicast datagrams it transmits.
4. The `setsockopt()` API uses the `IP_MULTICAST_IF` socket option, which defines the local interface over which the multicast datagrams are sent.
5. The `sendto()` API sends multicast datagrams to the specified group IP addresses.
6. The `close()` API closes any open socket descriptors.

Socket flow of events: Receiving multicast datagrams

The second example uses the following sequence of API calls:

1. The `socket()` API returns a socket descriptor representing an endpoint. The statement also identifies that the INET (Internet Protocol) address family with the TCP transport (`SOCK_DGRAM`) is used for this socket. This socket sends datagrams to another application.
2. The `setsockopt()` API sets the `SO_REUSEADDR` socket option to allow multiple applications to receive datagrams that are destined to the same local port number.
3. The `bind()` API specifies the local port number. In this example, the IP address is specified as `INADDR_ANY` to receive datagrams that are addressed to the multicast group.
4. The `setsockopt()` API uses the `IP_ADD_MEMBERSHIP` socket option, which joins the multicast group that receives the datagrams. When joining a group, specify the class D group address along with the IP address of a local interface. The system must call the `IP_ADD_MEMBERSHIP` socket option for each local interface that receives the multicast datagrams. In this example, the multicast group (225.1.1.1) is joined on the local 9.5.1.1 interface.

Note: The `IP_ADD_MEMBERSHIP` option must be called for each local interface over which the multicast datagrams are to be received.

5. The `read()` API reads multicast datagrams that are being sent.
6. The `close()` API closes any open socket descriptors.

Related concepts:

“IP multicasting” on page 61

IP multicasting allows an application to send a single IP datagram that a group of hosts in a network can receive.

Related reference:

“Example: Sending multicast datagrams”

This example enables a socket to send multicast datagrams.

Related information:

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`bind()`--Set Local Address for Socket API

`setsockopt()`--Set Socket Options API

`read()`--Read from Descriptor API

`sendto()`--Send Data API

Example: Sending multicast datagrams

This example enables a socket to send multicast datagrams.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
```

```
struct in_addr      localInterface;
struct sockaddr_in  groupSock;
int                 sd;
int                 datalen;
char                databuf[1024];
```

```

int main (int argc, char *argv[])
{
    /* -----*/
    /*      */
    /* Send Multicast Datagram code example.      */
    /*      */
    /* -----*/

    /*
     * Create a datagram socket on which to send.
     */
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sd < 0) {
        perror("opening datagram socket");
        exit(1);
    }

    /*
     * Initialize the group sockaddr structure with a
     * group address of 225.1.1.1 and port 5555.
     */
    memset((char *) &groupSock, 0, sizeof(groupSock));
    groupSock.sin_family = AF_INET;
    groupSock.sin_addr.s_addr = inet_addr("225.1.1.1");
    groupSock.sin_port = htons(5555);

    /*
     * Disable loopback so you do not receive your own datagrams.
     */
    {
        char loopch=0;

        if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_LOOP,
            (char *)&loopch, sizeof(loopch)) < 0) {
            perror("setting IP_MULTICAST_LOOP:");
            close(sd);
            exit(1);
        }
    }

    /*
     * Set local interface for outbound multicast datagrams.
     * The IP address specified must be associated with a local,
     * multicast-capable interface.
     */
    localInterface.s_addr = inet_addr("9.5.1.1");
    if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF,
        (char *)&localInterface,
        sizeof(localInterface)) < 0) {
        perror("setting local interface");
        exit(1);
    }

    /*
     * Send a message to the multicast group specified by the
     * groupSock sockaddr structure.
     */
    datalen = 10;
    if (sendto(sd, databuf, datalen, 0,
        (struct sockaddr*)&groupSock,
        sizeof(groupSock)) < 0)
    {
        perror("sending datagram message");
    }
}

```


Related reference:

“Examples: Using multicasting with AF_INET” on page 167

With IP multicasting, an application can send a single IP datagram that a group of hosts in a network can receive.

Example: Receiving multicast datagrams

This example enables a socket to receive multicast datagrams.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

struct sockaddr_in    localSock;
struct ip_mreq        group;
int                  sd;
int                  datalen;
char                 databuf[1024];

int main (int argc, char *argv[])
{
    /* ----- */
    /*                                     */
    /* Receive Multicast Datagram code example. */
    /*                                     */
    /* ----- */

    /*
     * Create a datagram socket on which to receive.
     */
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sd < 0) {
        perror("opening datagram socket");
        exit(1);
    }

    /*
     * Enable SO_REUSEADDR to allow multiple instances of this
     * application to receive copies of the multicast datagrams.
     */
    {
        int reuse=1;

        if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                       (char *)&reuse, sizeof(reuse)) < 0) {
            perror("setting SO_REUSEADDR");
            close(sd);
            exit(1);
        }
    }

    /*
     * Bind to the proper port number with the IP address
     * specified as INADDR_ANY.
     */
    memset((char *) &localSock, 0, sizeof(localSock));
    localSock.sin_family = AF_INET;
    localSock.sin_port = htons(5555);;
```

```

localSock.sin_addr.s_addr = INADDR_ANY;

if (bind(sd, (struct sockaddr*)&localSock, sizeof(localSock))) {
    perror("binding datagram socket");
    close(sd);
    exit(1);
}

/*
 * Join the multicast group 225.1.1.1 on the local 9.5.1.1
 * interface. Note that this IP_ADD_MEMBERSHIP option must be
 * called for each local interface over which the multicast
 * datagrams are to be received.
 */
group.imr_multiaddr.s_addr = inet_addr("225.1.1.1");
group.imr_interface.s_addr = inet_addr("9.5.1.1");
if (setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
               (char *)&group, sizeof(group)) < 0) {
    perror("adding multicast group");
    close(sd);
    exit(1);
}

/*
 * Read from the socket.
 */
datalen = sizeof(databuf);
if (read(sd, databuf, datalen) < 0) {
    perror("reading datagram message");
    close(sd);
    exit(1);
}
}

```

Example: Updating and querying DNS

This example shows how to query and update Domain Name System (DNS) records.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```

/*****
/* This program updates a DNS using a transaction signature (TSIG) to
/* sign the update packet. It then queries the DNS to verify success.
*****/

/*****
/* Header files needed for this sample program
*****/
#include <stdio.h>
#include <errno.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>

/*****
/* Declare update records - a zone record, a pre-requisite record, and
/* 2 update records
*****/
ns_updrec update_records[] =
{
    {
        {NULL,&update_records[1]},
        {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
},
{
    {&update_records[0],&update_records[2]},
    {&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
},
{
    {&update_records[1],&update_records[3]},
    {&update_records[1],&update_records[3]},
    ns_s_ud,          /* update record */
    "mypc.mydomain.ibm.com.",
    ns_c_in,
    ns_t_a,           /* IPv4 address */
    10,
    (unsigned char *)"10.10.10.10",
    11,
    ns_uop_add,       /* to be added */
    NULL,
    NULL,
    0
},
{
    {&update_records[2],NULL},
    {&update_records[2],NULL},
    ns_s_ud,          /* update record */
    "mypc.mydomain.ibm.com.",
    ns_c_in,
    ns_t_aaaa,        /* IPv6 address */
    10,
    (unsigned char *)"fedc:ba98:7654:3210:fedc:ba98:7654:3210",
    39,
    ns_uop_add,       /* to be added */
    NULL,
    NULL,
    0
}
};

/*****
/* These two structures define a key and secret that must match the one */
/* configured on the DNS : */
/* allow-update { */
/* key my-long-key.; */
/* } */
/*
/* This must be the binary equivalent of the base64 secret for */
/* the key */

```

```

/*****/
unsigned char secret[18] =
{
    0x6E,0x86,0xDC,0x7A,0xB9,0xE8,0x86,0x8B,0xAA,
    0x96,0x89,0xE1,0x91,0xEC,0xB3,0xD7,0x6D,0xF8
};

ns_tsig_key my_key = {
    "my-long-key",          /* This key must exist on the DNS */
    NS_TSIG_ALG_HMAC_MD5,
    secret,
    sizeof(secret)
};

void main()
{
    /*****/
    /* Variable and structure definitions. */
    /*****/
    struct state res;
    int result, update_size;
    unsigned char update_buffer[2048];
    unsigned char answer_buffer[2048];
    int buffer_length = sizeof(update_buffer);

    /* 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 */

    /* Build an update buffer (packet to be sent) from the update records */
    update_size = 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;

/* Send a TSIG signed update to the DNS */
    result = res_nsendsigned(&res, update_buffer, update_size,
                            &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()
/* can 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);
/*
*****/
/*****
/* Now verify that our update actually worked!
/* We choose to use TCP and not UDP, so set the appropriate option now
/* that the res variable has been initialized. We also want to ignore
/* the local cache and always send the query to the DNS server.
*****/

res.options |= RES_USEVC|RES_NOCACHE;

/* 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));
}
/*****
/* The output on a failure will be:
/*
/* query domain failed. result = -1
/* errno: 0: There is no error.
/* h_errno: 5: Unknown host
*****/
return;
}

```

Related concepts:

“Thread safety” on page 58

A function is considered threadsafe if you can start it simultaneously in multiple threads within the same process. A function is threadsafe only if all the functions it calls are also threadsafe. Socket APIs consist of system and network functions, which are both threadsafe.

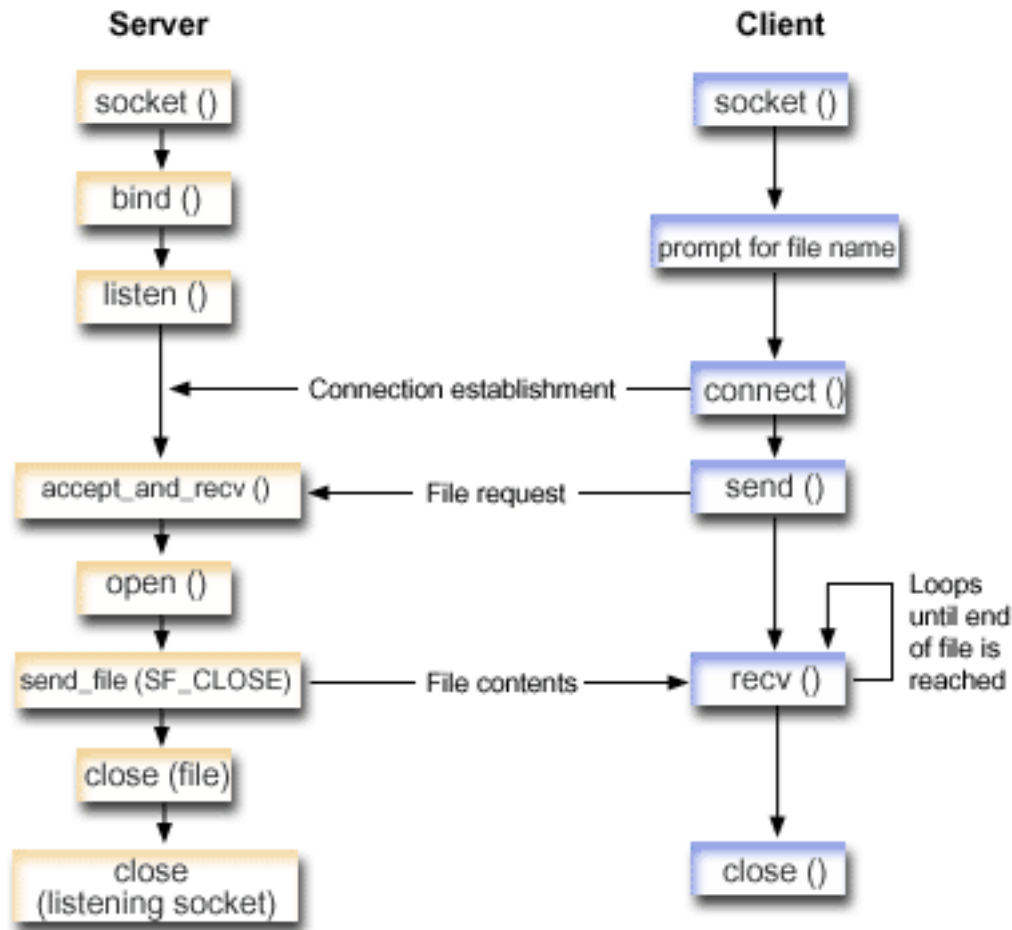
Related reference:

“Data caching” on page 67

Data caching of responses to Domain Name System (DNS) queries is done by IBM i sockets in an effort to lessen the amount of network traffic. The cache is added to and updated as needed.

Examples: Transferring file data using `send_file()` and `accept_and_recv()` APIs

These examples enable a server to communicate with a client by using the `send_file()` and `accept_and_recv()` APIs.



Socket flow of events: Server sends contents of a file

The following sequence of the socket calls provides a description of the graphic. It also describes the relationship between two applications that send and receive files. The first example uses the following sequence of API calls:

1. The server calls `socket()`, `bind()`, and `listen()` to create a listening socket.
2. The server initializes the local and remote address structures.
3. The server calls `accept_and_recv()` to wait for an incoming connection and to wait for the first data buffer to arrive over this connection. This call returns the number of bytes that is received and the local and remote addresses that are associated with this connection. This call is a combination of the `accept()`, `getsockname()`, and `recv()` APIs.
4. The server calls `open()` to open the file whose name was obtained as data on the `accept_and_recv()` from the client application.
5. The `memset()` API is used to set all of the fields of the `sf_parms` structure to an initial value of 0. The server sets the file descriptor field to the value that the `open()` API returned. The server then sets the

file bytes field to -1 to indicate that the server should send the entire file. The system is sending the entire file, so you do not need to assign the file offset field.

6. The server calls the `send_file()` API to transmit the contents of the file. The `send_file()` API does not complete until the entire file has been sent or an interruption occurs. The `send_file()` API is more efficient because the application does not need to go into a `read()` and `send()` loop until the file finishes.
7. The server specifies the `SF_CLOSE` flag on the `send_file()` API. The `SF_CLOSE` flag informs the `send_file()` API that it should automatically close the socket connection when the last byte of the file and the trailer buffer (if specified) have been sent successfully. The application does not need to call `close()` if the `SF_CLOSE` flag is specified.

Socket flow of events: Client request for file

The second example uses the following sequence of API calls:

1. This client program takes from zero to two parameters.
The first parameter (if specified) is the dotted-decimal IP address or the host name where the server application is located.
The second parameter (if specified) is the name of the file that the client attempts to obtain from the server. A server application sends the contents of the specified file to the client. If the user does not specify any parameters, then the client uses `INADDR_ANY` for the server's IP address. If the user does not specify a second parameter, the program prompts the user to enter a file name.
2. The client calls `socket()` to create a socket descriptor.
3. The client calls `connect()` to establish a connection to the server. Step one obtained the IP address of the server.
4. The client calls `send()` to inform the server what file name it wants to obtain. Step one obtained the name of the file.
5. The client goes into a "do" loop calling `recv()` until the end of the file is reached. A return code of 0 on the `recv()` means that the server closed the connection.
6. The client calls `close()` to close the socket.

Related concepts:

"File data transfer—`send_file()` and `accept_and_recv()`" on page 62

IBM i sockets provide the `send_file()` and `accept_and_recv()` APIs that enable faster and easier file transfers over connected sockets.

Related information:

`accept()`--Wait for Connection Request and Make Connection API

`recv()`--Receive Data API

`send()`--Send Data API

`listen()`--Invite Incoming Connections Requests API

`close()`--Close File or Socket Descriptor API

`socket()`--Create Socket API

`bind()`--Set Local Address for Socket API

`getsockname()`--Retrieve Local Address of Socket API

`open()`--Open File API

`read()`--Read from Descriptor API

`connect()`--Establish Connection or Destination Address API

Example: Using `accept_and_recv()` and `send_file()` APIs to send contents of a file

This example enables a server to communicate with a client by using the `send_file()` and `accept_and_recv()` APIs.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* Server example send file data to client */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
    int    i, num, rc, flag = 1;
    int    fd, listen_sd, accept_sd = -1;

    size_t local_addr_length;
    size_t remote_addr_length;
    size_t total_sent;

    struct sockaddr_in6  addr;
    struct sockaddr_in6  local_addr;
    struct sockaddr_in6  remote_addr;
    struct sf_parms      parms;

    char    buffer[255];

    /* If an argument is specified, use it to
     * control the number of incoming connections
     */
    if (argc >= 2)
        num = atoi(argv[1]);
    else
        num = 1;

    /* Create an AF_INET6 stream socket to receive
     * incoming connections on
     */
    listen_sd = socket(AF_INET6, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
        perror("socket() failed");
        exit(-1);
    }

    /* Set the SO_REUSEADDR bit so that you do not
     * need to wait 2 minutes before restarting
     * the server
     */
    rc = setsockopt(listen_sd,
                     SOL_SOCKET,
                     SO_REUSEADDR,
                     (char *)&flag,
                     sizeof(flag));

    if (rc < 0)
    {
        perror("setsockopt() failed");
        close(listen_sd);
        exit(-1);
    }
}
```



```

}

/*****/
/* Bind the socket */
/*****/
memset(&addr, 0, sizeof(addr));
addr.sin6_family = AF_INET6;
memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
addr.sin6_port = htons(SERVER_PORT);
rc = bind(listen_sd,
          (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
    perror("bind() failed");
    close(listen_sd);
    exit(-1);
}

/*****/
/* Set the listen backlog */
/*****/
rc = listen(listen_sd, 5);
if (rc < 0)
{
    perror("listen() failed");
    close(listen_sd);
    exit(-1);
}

/*****/
/* Initialize the local and remote addr lengths */
/*****/
local_addr_length = sizeof(local_addr);
remote_addr_length = sizeof(remote_addr);

/*****/
/* Inform the user that the server is ready */
/*****/
printf("The server is ready\n");

/*****/
/* Go through the loop once for each connection */
/*****/
for (i=0; i < num; i++)
{
    /*****/
    /* Wait for an incoming connection */
    /*****/
    printf("Iteration: %d\n", i+1);
    printf(" waiting on accept_and_recv()\n");

    rc = accept_and_recv(listen_sd,
                        &accept_sd,
                        (struct sockaddr *)&remote_addr,
                        &remote_addr_length,
                        (struct sockaddr *)&local_addr,
                        &local_addr_length,
                        &buffer,
                        sizeof(buffer));

    if (rc < 0)
    {
        perror("accept_and_recv() failed");
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }

    printf(" Request for file: %s\n", buffer);
}

```

```

/*****
/* Open the file to retrieve */
/*****
fd = open(buffer, O_RDONLY);
if (fd < 0)
{
    perror("open() failed");
    close(listen_sd);
    close(accept_sd);
    exit(-1);
}

/*****
/* Initialize the sf_parms structure */
/*****
memset(&parms, 0, sizeof(parms));
parms.file_descriptor = fd;
parms.file_bytes      = -1;

/*****
/* Initialize the counter of the total number */
/* of bytes sent */
/*****
total_sent = 0;

/*****
/* Loop until the entire file has been sent */
/*****
do
{
    rc = send_file(&accept_sd, &parms, SF_CLOSE);
    if (rc < 0)
    {
        perror("send_file() failed");
        close(fd);
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }
    total_sent += parms.bytes_sent;
} while (rc == 1);

printf(" Total number of bytes sent: %d\n", total_sent);

/*****
/* Close the file that is sent out */
/*****
close(fd);
}

/*****
/* Close the listen socket */
/*****
close(listen_sd);

/*****
/* Close the accept socket */
/*****
if (accept_sd != -1)
    close(accept_sd);
}

```

Related information:

send_file()--Send a File over a Socket Connection API

accept_and_recv()

Example: Client request for a file

This example enables a client to request a file from the server and to wait for the server to send the contents of that file back.

Note: By using the examples, you agree to the terms of the “Code license and disclaimer information” on page 193.

```
/* ***** */
/* Client example requests file data from server */
/* ***** */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
    int    rc, sockfd;

    char    filename[256];
    char    buffer[32 * 1024];

    struct sockaddr_in6  addr;
    struct addrinfo hints, *res;

    /* ***** */
    /* Initialize the socket address structure */
    /* ***** */
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family = AF_INET6;
    addr.sin6_port = htons(SERVER_PORT);

    /* ***** */
    /* Determine the host name and IP address of the */
    /* machine the server is running on */
    /* ***** */
    if (argc < 2)
    {
        memcpy(&addr.sin6_addr, &in6addr_any, sizeof(in6addr_any));
    }
    else if ((isdigit(*argv[1])) || (*argv[1] == ':'))
    {
        rc = inet_pton(AF_INET6, argv[1], &addr.sin6_addr.s6_addr);
    }
    else
    {
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET6;
        hints.ai_flags = AI_V4MAPPED;
        rc = getaddrinfo(argv[1], NULL, &hints, &res);
        if (rc != 0)
        {
            printf("Host not found! (%s)\n", argv[1]);
            exit(-1);
        }

        memcpy(&addr.sin6_addr,
               (&((struct sockaddr_in6 *) (res->ai_addr))->sin6_addr),
               sizeof(addr.sin6_addr));

        freeaddrinfo(res);
    }
}
```

```

}

/*****
/* Check to see if the user specified a file name */
/* on the command line */
*****/
if (argc == 3)
{
    strcpy(filename, argv[2]);
}
else
{
    printf("Enter the name of the file:\n");
    gets(filename);
}

/*****
/* Create an AF_INET6 stream socket */
*****/
sockfd = socket(AF_INET6, SOCK_STREAM, 0);
if (sockfd < 0)
{
    perror("socket() failed");
    exit(-1);
}
printf("Socket completed.\n");

/*****
/* Connect to the server */
*****/
rc = connect(sockfd,
             (struct sockaddr *)&addr,
             sizeof(struct sockaddr_in6));
if (rc < 0)
{
    perror("connect() failed");
    close(sockfd);
    exit(-1);
}
printf("Connect completed.\n");

/*****
/* Send the request over to the server */
*****/
rc = send(sockfd, filename, strlen(filename) + 1, 0);
if (rc < 0)
{
    perror("send() failed");
    close(sockfd);
    exit(-1);
}
printf("Request for %s sent\n", filename);

/*****
/* Receive the file from the server */
*****/
do
{
    rc = recv(sockfd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
        perror("recv() failed");
        close(sockfd);
        exit(-1);
    }
    else if (rc == 0)
    {

```

```

        printf("End of file\n");
        break;
    }
    printf("%d bytes received\n", rc);
} while (rc > 0);

/*****
/* Close the socket
*****/
close(sockfd);
}

```

Xsockets tool

The Xsockets tool is one of the many tools that is supplied with the IBM i product. All tools are stored in the QUSRTOOL library. The Xsockets tool allows programmers to interactively work with socket APIs.

The Xsockets tool allows you to do the following tasks:

- Learn about the socket APIs.
- Re-create specific scenarios interactively to help debug.

Note: The Xsockets tool is supplied in an as-is format.

Prerequisites for using Xsockets

Do the following tasks before using Xsockets:

- Install the ILE C language.
- Install the System Openness Includes feature (option 13) of the IBM i licensed program.
- Install the IBM HTTP Server for i (5770-DG1) licensed program.

Note: This is needed if you plan to use Xsockets in a Web browser.

- The IBM Developer Kit for Java (5770-JV1) licensed program is installed.

Note: This is needed if you plan to use Xsockets in a Web browser.

Configuring Xsockets

The Xsockets tool is available in two versions. The first version is integrated with the IBM i client. The integrated version is completely created by the first set of instructions. The second version uses a Web browser as the client.

If you want to use the Web browser client, you must complete setup instructions for the integrated version first.

To create the Xsockets tool, complete the following steps:

1. To unpack the tool, enter

```
CALL QUSRTOOL/UNPACKAGE ('*ALL' ' 1)
```

on a command line.

Note: You must have 10 characters between the opening and closing single quotation marks (').

2. To add the QUSRTOOL library to your library list, enter

```
ADDLIB QUSRTOOL
```

on a command line.

3. Create a library in which to create the Xsocket program files by entering

```
CRTLIB <library-name>
```

on a command line. The <library-name> is the the library in which you want the Xsockets tool objects created. For example,

```
CRTLIB MYXSOCKET
```

is a valid library name.

Note: If XSOCKETS is used as the library, you will be able to skip a configuration step when configuring Xsockets for the Web. Do not add Xsockets tool objects to the QUSRTOOL library. Adding Xsockets tool objects to the QUSRTOOL library can interfere with the use of other tools within that directory.

4. To add this library to the library list, enter ADDLIB <library-name> on the command line. The <library-name> is the library that you created in step 3. For example, if MYXSOCKET was used as the library name, then ADDLIB MYXSOCKET must be entered.
5. Create the installation program TSOCRT that automatically installs the Xsockets tool by entering: CRTCLPGM <library-name>/TSOCRT QUSRTOOL/QATTCL on the command line.
6. To call the installation program, enter:

```
CALL TSOCRT library-name
```

on the command line. In the place of library-name, use the library you created in step 3. For example, to create the tool in the MYXSOCKET library, enter:

```
CALL TSOCRT MYXSOCKET
```

Note: This might take a few minutes to complete.

If you do not have job control (*JOBCTL) special authority when you call TSOCRT to create the sockets tool, the givedescriptor() socket function returns errors when an attempt is made to pass a descriptor to a job that is not the one you are running.

TSOCRT creates a CL program, an ILE C program (two modules are created), two ILE C service programs (two modules are created), and three display files. Whenever you want to use the tool, you must add the library to your library list. All objects created by the tool have a name that is prefixed by TSO.

Note: The integrated version does not support GSKit secure socket APIs. If you want to write socket programs that use the integrated APIs, you should use the browser-based version of the tool.

Related concepts:

“Using Xsockets” on page 189

You can work with the Xsockets tool either from the integrated client or from a Web browser.

Related tasks:

“Using integrated Xsockets” on page 189

Follow these instructions to use the Xsockets tool on an integrated client.

“Updating configuration files” on page 187

After you have installed the integrated Xsockets tool, you must complete manual changes to several configuration files for the instance.

“Using Xsockets in a Web browser” on page 190

Follow these instructions for using the Xsockets tool in a Web browser.

What is created by integrated Xsocket setup

This table lists the objects created by the installation program. All of the created objects reside in the specified library.

Table 19. Objects created during Xsocket installation

Object name	Member name	Source file name	Object type	Extension	Description
TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	Module that is used for interfacing between JSP and TSOSTSOC
TSODLT	TSODLT	QATTCL	*PGM	CLP	CL program to delete the tool objects, the source file members, or both.
TSOXSOCK	N/A	N/A	*PGM	C	Main program that is used for the SOCKETS interactive tool.
TSOXGJOB	N/A	N/A	*SRVPGM	C	Service program that is used in support of the SOCKETS interactive tool
TSOJNI	N/A	N/A	*SRVPGM	C	Service program that is used for interfacing between JSP and TSOSTSOC in support of the SOCKETS interactive tool.
TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	Module that is used in the creation of the TSOXSOCK program. The source file contains the main() routine.
TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	Module that is used in the creation of the TSOXSOCK program. The source file contains the routines that actually call the socket functions.
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	Module that is used in the creation of the TSOXGJOB service program. The source file contains the routine that identifies the internal job. This internal job identifier is made up of the job name, user ID, and job number.
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	Display file used by the Xsockets tool for the main window that contains the sockets functions.

Table 19. Objects created during Xsocket installation (continued)

Object name	Member name	Source file name	Object type	Extension	Description
TSOFUN	TDSOFUN	QATTDDS	*FILE	DSPF	Display file used by the XSockets tool in support of the various socket functions.
TSOMNU	TDSOMNU	QATTDDS	*FILE	DSPF	Display file used by the XSockets tool that supports the menu bar.
QATTIFS2	N/A	N/A	*FILE	PF-DTA	Contains the JAR file used by the Lightweight Web Infrastructure.

Configuring Xsockets to use a Web browser

You can configure the Xsockets tool to allow access through a Web browser. You can implement these instructions multiple times on the same system to create different server instances. With multiple instances, you can run multiple versions at the same time on different listening ports.

Related concepts:

“Using Xsockets” on page 189

You can work with the Xsockets tool either from the integrated client or from a Web browser.

Related tasks:

“Using Xsockets in a Web browser” on page 190

Follow these instructions for using the Xsockets tool in a Web browser.

Configuring an Integrated Web Application Server

To use the Xsockets tool in a Web browser, you need to configure an integrated Web application server.

Before configuring a Web browser to work with the Xsockets tool, you must first configure Xsockets. See Configuring Xsockets to learn how to do this.

1. Verify the HTTP admin instance is running under the QHTTPSVR subsystem. You can start it with the following CL command if it is not running:

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
```

2. In a Web browser, enter:

```
http://<system_name>:2001/.
```

where <system_name> is the machine name of the system. For example: http://mysystemi:2001/.

3. On the IBM i Tasks page, select **IBM Web Administration for i**.
4. From the top menu, select the **Setup** tab.
5. Click **Create Application Server**.
6. Under integrated Web application server, select the type of application server to create, and click **Next**.
7. Enter the name for the server instance, and click **Next**. For example, if this instance serves the Xsockets tool in a browser, then you can use the name xsocket. A new HTTP Server (powered by Apache) will be created in addition to the integrated Web application server.

Note: Use the default HTTP Server name and description.

8. Select a range for internal ports to be used by the application server and click **Next**.

Note: Use a port number that is greater than 1024.

9. Select the IP address, an available port that you want to use, and click **Next**.

Note: Use a port number that is greater than 1024. Do not select the default port number 80.

10. Click **Next** to use the default value for specifying the user ID.
11. Click **Finish** to confirm the Application Server and HTTP Server (powered by Apache) configuration settings.

Related tasks:

“Updating configuration files”

After you have installed the integrated Xsockets tool, you must complete manual changes to several configuration files for the instance.

“Testing Xsockets tool in a Web browser” on page 189

After you have completed configuring the Xsockets Web application, you are ready to test the Xsockets tool within a browser. The server and application instance should already be started.

“Using Xsockets in a Web browser” on page 190

Follow these instructions for using the Xsockets tool in a Web browser.

Updating configuration files

After you have installed the integrated Xsockets tool, you must complete manual changes to several configuration files for the instance.

You need to update these files: the JAR file, the web.xml file, and the httpd.conf file.

1. Copy the JAR file From a command line, enter this command:

```
CPY OBJ('/QSYS.LIB/XXXX.LIB/QATTIFS2.FILE/XSOCK.MBR')  
TOOBJ('/www/<server_name>/xsock.war') FROMCCSID(*OBJ) TOCCSID(819) OWNER(*NEW)
```

where XXXX is the library name you created during Xsockets configuration and <server_name> is the name of the server instance you created during Apache configuration. This is the integrated file system directory where you would like to store the XSocket JAR file.

2. Optional: Update the web.xml file:

Note: This step is only necessary if Xsockets was installed to a library other than XSOCKETS during Xsockets configuration.

- a. From a command line, enter

```
CD DIR('/www/<server_name>')
```

where <server_name> is the name of the server instance you created during Apache configuration.

- b. From a command line, enter

```
STRQSH CMD('jar xf xsock.war')
```

to extract the configuration files stored in the XSocket JAR file.

- c. From a command line, enter

```
wrklnk 'WEB-INF/web.xml'
```

- d. Press function 2 (Edit) to edit the file.

- e. Find the </servlet-class> line in the web.xml file.

- f. Update the following code after this line:

```
<init-param>  
    <param-name>library</param-name>  
    <param-value>xsockets</param-value>  
</init-param>
```

In place of the *xsockets*, insert the library name that you created during Xsockets configuration.

g. Save the file and exit the edit session.

h. From a command line, enter

```
STRQSH CMD('jar cmf META-INF/MANIFEST.MF xsock.war lib WEB-INF')
```

to create a new XSocketS JAR file containing the updated configuration file.

3. Optional: Add the authority check to httpd.conf file. This forces Apache to authenticate users trying to access the XsocketS Web application.

Note: It is also necessary for getting write access to create UNIX sockets.

a. From a command line, enter

```
wrklnk '/www/<server_name>/conf/httpd.conf'
```

where *<server_name>* is the name of the server instance you created during the Apache configuration. For example, if you choose xsocks for the server name, you can enter:

```
wrklnk '/www/xsocks/conf/httpd.conf'
```

b. Press function 2 (Edit) to edit the file.

c. Insert the following lines at the end of the file.

```
<Location /xsock>
  AuthName "X Socket"
  AuthType Basic
  PasswdFile %%SYSTEM%%
  UserId %%CLIENT%%
  Require valid-user
  order allow,deny
  allow from all
</Location>
```

d. Save the file and exit the edit session.

Related tasks:

“Configuring an Integrated Web Application Server” on page 186

To use the XsocketS tool in a Web browser, you need to configure an integrated Web application server.

“Configuring XsocketS” on page 183

The XsocketS tool is available in two versions. The first version is integrated with the IBM i client. The integrated version is completely created by the first set of instructions. The second version uses a Web browser as the client.

“Configuring XsocketS Web application”

After you have configured the integrated Web application server and HTTP Server (powered by Apache) server instance, you must configure a new application to use the XsocketS tool in a Web browser.

Configuring XsocketS Web application

After you have configured the integrated Web application server and HTTP Server (powered by Apache) server instance, you must configure a new application to use the XsocketS tool in a Web browser.

1. Under **Manage**, select the Application Server that you have created.
2. Under **Application Server Wizards** in the left pane, select **Install New Application**.
3. Specify the location of the JAR file that contains the application, and click **Next**. This is the JAR file that was created from '/QSYS.LIB/XXX.LIB/QATTIFS2.FILE/XSOCK.MBR' and was updated when you updated the configuration files.
4. Enter the name for the application and accept the default value for the context root. Click **Next**. For example, if this application serves the XsocketS tool in a browser, you can use XSocketS.
5. Click **Finish** to complete the application configuration for the XsocketS tool.

Related tasks:

“Updating configuration files” on page 187

After you have installed the integrated XsocketS tool, you must complete manual changes to several

configuration files for the instance.

Testing Xsockets tool in a Web browser

After you have completed configuring the Xsockets Web application, you are ready to test the Xsockets tool within a browser. The server and application instance should already be started.

1. If the server and application instance is not already started, start the server instance with the following command on a command line:

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(<server_name>)
```

where *<server_name>* is the name of the server instance you created during the HTTP Server (powered by Apache) configuration. This takes a while.

2. Check the status of the server by issuing the Work with Active Jobs (WRKACTJOB) command from the command line interface. You should see one job with your *server_name*, PGM-QLWISVR Function, with JVAW status, and all additional jobs should have the SIGW status. If this is the case, then you can proceed to the next step.
3. In a browser, enter the following URL:

```
http://<system_name>:<port>/xsock/index
```

where *<system_name>* is the machine name of the system and *<port>* is the port number that you chose during the Apache configuration.

4. When prompted, enter your user name and password for the server. The Web client for Xsocket should appear.

Related tasks:

“Configuring an Integrated Web Application Server” on page 186

To use the Xsockets tool in a Web browser, you need to configure an integrated Web application server.

Using Xsockets

You can work with the Xsockets tool either from the integrated client or from a Web browser.

To work with an integrated version of Xsockets, you must configure the Xsockets tool. In addition to configuring the Xsockets tool for an integrated client, you must also complete the steps in Configuring Xsockets to use a Web browser if you prefer to work with the tool in a browser environment. Many of the concepts are similar between the two versions of the tools. Both tools allow you to issue socket calls interactively and both tools provide errors for issued socket calls; however, the interfaces do have some differences.

Note: If you want to work with socket programs that use the GSKit secure socket APIs, you must use the Web version of the tool.

Related concepts:

“Configuring Xsockets to use a Web browser” on page 186

You can configure the Xsockets tool to allow access through a Web browser. You can implement these instructions multiple times on the same system to create different server instances. With multiple instances, you can run multiple versions at the same time on different listening ports.

Related tasks:

“Configuring Xsockets” on page 183

The Xsockets tool is available in two versions. The first version is integrated with the IBM i client. The integrated version is completely created by the first set of instructions. The second version uses a Web browser as the client.

Using integrated Xsockets

Follow these instructions to use the Xsockets tool on an integrated client.

1. From a command line, add the library in which the Xsockets tool exists to your library list by issuing this command:

```
ADDLIBLE <library-name>
```

where the <library-name> is the name of the library you created during integrated Xsockets configuration. For example, if the name of the library is MYXSOCKET, then enter:

```
ADDLIBLE MYXSOCKET
```

2. On a command line interface, enter:

```
CALL TSOXSOCK
```
3. From the Xsocket window that is shown, you can access all socket routines through its menu bar and selection field. This window is always shown after you choose a socket API. You can use this interface to select socket programs that already exist. To work with a new socket, follow these steps:
 - a. In the list of socket APIs, select **socket** and press Enter.
 - b. In the **socket() prompt** window that displays, select the appropriate Address Family, Socket Type, and Protocol for the socket, and press Enter.
 - c. Select **Descriptor** and select **Select descriptor**.

Note: If other socket descriptors already exist, this displays a list of active socket descriptors.

- d. From the list that displays, select the socket descriptor that you created.

Note: If other socket descriptors exist, the tool automatically applies a socket API to the latest socket descriptor.

4. From the list of socket APIs, select a socket API with which you want to work, whatever socket descriptor you chose in step 3c is used on that socket API. As soon as you select a socket API, a series of windows are displayed where you can provide specific information about the socket API. For example, if you select connect(), you need to provide the address length, address family, and address data in the resulting windows. The socket API chosen is then called with this information that you provided. Any errors that occur on a socket API are displayed back to the user as an errno.

Notes:

1. The Xsockets tool uses the graphical support for DDS. Thus, how data is entered and how selections are made from the windows you see depend on whether you are using a graphical display station or a nongraphical display station. For example, on a graphical display station, you can see the selection field for the socket APIs as a check box; otherwise, you might see a single field.
2. Be aware that there are ioctl() requests that are available on a socket which have not been implemented in the tool.

Using Xsockets in a Web browser

Follow these instructions for using the Xsockets tool in a Web browser.

Ensure that you have completed all the Xsockets configuration and all the necessary Web browser configuration before working with the Xsockets tool in a Web browser. Also ensure that cookies are enabled.

1. In a Web browser, type:

```
http://system-name:2001/
```

where *system-name* is the name of the system that contains the server instance.

2. Select **Administration**.
3. From the left navigation, select **Manage HTTP Servers**.
4. Select your instance name, and click **Start**. You can also start the server instance from a command line by entering:

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(<instance_name>)
```

where *<instance_name>* is the name of your HTTP Server created in the Apache configuration. For example, you can use the server instance name `xsocks`.

5. To access the Xsockets Web application, enter this URL in a browser:

`http://<system_name>:<port>/xsock/index`

where *<system_name>* is the machine name of the system and *<port>* is the port specified when you created the HTTP instance. For example, if the system name is `mySystem1` and the HTTP Server instance listens on port 1025, you can enter:

`http://mySystem1:1025/xsock/index`

6. After the Xsockets tool loads in the Web browser, you can work with the existing socket descriptor or create a new one. To create a new socket descriptor, follow these steps:
 - a. From the **Xsocket Menu**, select **socket**.
 - b. In the **Xsocket Query** window that displays, select the appropriate Address Family, Socket Type, and Protocol for this socket descriptor. Click **Submit**. As soon as the page reloads, the new socket descriptor is displayed in the **Socket** pull-down menu.
 - c. From the **Xsocket Menu**, select API calls to which you want to apply this socket descriptor. As with the integrated version of the Xsockets tool, the tool automatically applies API calls to the latest socket descriptor if you do not select a socket descriptor.

Related concepts:

“Configuring Xsockets to use a Web browser” on page 186

You can configure the Xsockets tool to allow access through a Web browser. You can implement these instructions multiple times on the same system to create different server instances. With multiple instances, you can run multiple versions at the same time on different listening ports.

Related tasks:

“Configuring Xsockets” on page 183

The Xsockets tool is available in two versions. The first version is integrated with the IBM i client. The integrated version is completely created by the first set of instructions. The second version uses a Web browser as the client.

“Configuring an Integrated Web Application Server” on page 186

To use the Xsockets tool in a Web browser, you need to configure an integrated Web application server.

Deleting objects created by the Xsockets tool

You might need to delete objects that are created by the Xsockets tool. The program named `TSODLT` is created by the installation program to remove the objects created by the tool (except the library and the program `TSODLT`) or to remove the source members used by the Xsockets tool, or both.

The following set of commands allow you to delete these objects:

To delete **ONLY** the source members used by the tool, enter the following command :

```
CALL TSODLT (*YES *NONE)
```

To delete **ONLY** objects that the tool creates, enter the following command:.

```
CALL TSODLT (*NO library-name)
```

To delete **BOTH** source members and objects created by the tool, enter the following command:

```
CALL TSODLT (*YES library-name)
```

Customizing Xsockets

You can change the Xsockets tool by adding additional support for the socket network routines (such as `inet_addr()`).

If you choose to customize this tool to meet your own needs, it is recommended that you do not make changes in the QUSRTOOL library. Instead, copy the source files into a separate library and make the changes there. This preserves the original files in the QUSRTOOL library so they are available if needed in the future. You can use the TSOCRT program to recompile the tool after making your changes (note that if the source files are copied to a separate library, you also need to make changes in TSOCRT to use it). Use the TSODLT program to remove old versions of the tool objects before creating the tool.

Serviceability tools

Because the use of sockets and secure sockets continues to grow to accommodate e-business applications and servers, the current serviceability tools need to keep up with this demand.

Enhanced serviceability tools help you complete traces on socket programs to find solutions to errors within socket and SSL-enabled applications. These tools help you and support center personnel to determine where socket problems are by selecting socket traits, such as IP address or port information.

The following table gives an overview for the each of these service tools.

Table 20. Serviceability tools for socket and secure sockets

Serviceability tool	Description
Licensed Internal Code trace filtering (TRCINT and TRCCNN)	Provides selective trace on sockets. You can restrict sockets trace on address family, socket type, protocol, IP address, and port information. You can also limit traces to only certain categories of socket APIs and also to only those sockets that have the SO_DEBUG socket option set. A Licensed Internal Code trace can be filtered by thread, task, user profile, job name, or server name.
Trace job with STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR))	STRTRC command provides additional parameters that generate output that is separated from all other non-socket related trace points. This output contains return code and errno information when an error is encountered during a socket operation.
Flight recorder tracing	Sockets Licensed Internal Code component traces include a dump of the flight recorder entries for each socket operation performed.
Associated job information	Allows service personnel and programmers to find all jobs that are associated to a connected or listening socket. This information can be viewed using NETSTAT for those socket applications using an address family of AF_INET or AF_INET6.
NETSTAT connection status(option 3) to enable SO_DEBUG	Provides enhanced low-level debug information when the SO_DEBUG socket option is set on a socket application.
Secure socket return code and message processing	Presents standardized secure socket return code messages through gsk_strerror() for GSKit APIs. SSL_strerror() and SSL_Perror() provide similar function for SSL_ APIs. There is also the hstrerror() API that provides return code information from resolver routines.
Performance data collection tracepoints	Provides a trace for the data flow from an application through sockets and the TCP/IP stack.

Related information:

SSL_strerror()--Retrieve SSL Runtime Error Message API

SSL_Perror()--Print SSL Error Message API

gsk_strerror()--Retrieve GSKit runtime error message API

hstrerror()--Retrieve Resolver Error Message API
Start Trace (STRTRC) command

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement 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.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

Programming interface information

This Socket programming publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

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.



Product Number: 5770-SS1

Printed in USA